ProDev™ WorkShop: Tester User's Guide

007–3986–004

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

Record of Revision

Version	Description
001	April 1999 Initial release as a separate product. Supports ProDev WorkShop 2.8 release.
002	June 2001 Supports ProDev WorkShop 2.9 release.
003	November 2001 Supports ProDev WorkShop 2.9.1 release.
004	September 2002 Released with the ProDev WorkShop 2.9.2 release.

Contents

Figures

Tables

Examples

Procedures

About This Guide

This publication documents the ProDev WorkShop Tester release 2.9.2 running on IRIX systems. WorkShop Tester is a UNIX-based software quality assurance toolset for dynamic test coverage over any set of tests. This product is intended for software and test engineers and their managers involved in the development, test, and maintenance of long-lived software projects.

Related Publications

The following documents contain additional information that may be helpful:

- *ProDev WorkShop: Debugger User's Guide*
- *ProDev WorkShop: Debugger Reference Manual*
- *ProDev WorkShop: Overview*
- *C Language Reference Manual*
- *MIPSpro C++ Programmer's Guide*
- *MIPSpro C and C++ Pragmas*
- *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*

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: `http://docs.sgi.com`. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man` *title* on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
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, contact SGI. 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 SGI 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 Web page:

  `http://docs.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 Parkway, M/S 535
  Mountain View, California 94043–1351

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

SGI values your comments and will respond to them promptly.

Using Tester

This chapter describes the Tester usage model. It shows the general approach of applying Tester for coverage analysis. It contains these sections:

- "Tester Overview", page 1, describes Tester and its capabilities.
- "Usage Model", page 5, describes the steps in testing, and using scripts to automate testing.

See the Glossary, page 115, for a list of commonly used terms and their meanings.

Tester Overview

WorkShop Tester is a quality assurance toolset for dynamic test coverage over sets of tests. The term *coverage* means that the test has executed a particular unit of source code.

In this product, *units* are functions, individual source lines, arcs, blocks, or branches. If the unit is a branch, covered means it has been executed under both true and false conditions. This product is intended for software and test engineers and their managers involved in the development, test, and maintenance of long-lived software projects.

WorkShop Tester provides these general benefits:

- Provides visualization of coverage data, which yields immediate insight into quality issues at both engineering and management levels
- Provides useful measures of test coverage over a set of tests/experiments
- Lets you view the coverage results of a dynamically shared object (DSO) by executables that use it
- Provides comparison of coverage over different program versions
- Provides tracing capabilities for function arcs that go beyond traditional test coverage tools
- Supports programs written in C, C++, and Fortran
- Is integrated into the CASEVision family of products

- Allows users to build and maintain higher quality software products

There are two versions of Tester:

- `cvcov` is the command line version of the test coverage program.
- `cvxcov` is the GUI version of the test coverage program.

Most of the functionality is available from either program, although the graphical representations of the data are available only from `cvxcov`, the GUI tool.

Test Coverage Data

Tester provides the following basic coverage:

- Basic block: how many times was this basic block executed?
- Function: how many times was this function executed?
- Branch: did this condition take on both TRUE and FALSE values?

You can also request the following coverage information:

- Arc: was function `F` called by function `A` and function `B`? Which arcs for function `F` were **not** taken?
- Source line coverage: how many times has this source line been executed and what percentage of source lines is covered?
- When the target program `execs`, `forks`, or `sprocs` another program, only the main target is tested, unless you specify which executables are to be tested, the parent and/or child programs.

Note: When you compile with the `-g` flag, you may create assembly blocks and branches that can never be executed, thus preventing "full" coverage from being achieved. These are usually negligible. However, if you compile with the `01` flag (the default), you can increase the number of executable blocks and branches.

Types of Experiments

You can conduct Tester coverage experiments for:

- Separate tests

- A set of tests operating on the same executable
- A list of executables related by `fork`, `exec`, or `sproc` commands
- A test group of executables sharing a common dynamically shared object (DSO)

Experiment Results

Tester presents the experiment results in these reports:

- Summary of test coverage, including user parameterized dynamic coverage metric
- List of functions, which can be sorted by count, file, or function name and filtered by percentage of block, branch, or function covered
- Comparison of test coverage between different versions of the same program
- Source or assembly code listing annotated with coverage data
- Breakdown of coverage according to contribution by tests within a test set or test group

The graphical user interface lets you view test results in different contexts to make them more meaningful. It provides:

- Annotated function call graph highlighting coverage by counts and percentage (ASCII function call graph supported as well)
- Annotated Source View showing coverage at the source language level
- Annotated Disassembly View showing coverage at the assembly language level
- Bar chart summary showing coverage by functions, lines, blocks, branches, and arcs

Multiple Tests

Tester supports multiple tests. You can:

- Define and run a test set to cover the same program.
- Define and run a test group to cover programs sharing a common DSO. This approach is useful if you want to test different client programs that bind with the same libraries.

- Automate test execution via command line interface as well as GUI mode.

Test Components

Each test is a named object containing the following:

- Instrumentation file: This describes the data to be collected.
- Executable: This is the program being instrumented for coverage analysis.
- Executable list: If the program you are testing can `fork`, `exec`, or `sproc` other executables and you want these other executables included in the test, then you can specify a list of executables for this purpose.
- Command: This defines the program.
- Instrumentation directory: The instrumentation directory contains directories representing different versions of the instrumented program and related data. Instrumentation directories are named `ver##<n>` where `n` is the version number. Several tests can share the same instrumentation directory. This is true for tests with the same instrumentation file and program version. The instrumentation directory contains the following files, which are automatically generated:

  ```
  <program|DSO>.Log     instrumentation log file (cvinstr)
  <program|DSO>.pixie   instrumented executable
  ```

  As part of instrumentation, you can filter the functions to be included or excluded in your test, through the directives `INCLUDE`, `EXCLUDE`, and `CONSTRAIN`.

- Experiment results: Test run coverage results are deposited in a results directory. Results directories are named `exp##<`*n*`>` where *n* corresponds to the instrumentation directory used in the experiment. There is one results directory for each version of the program in the instrumentation directory for this test. Note that results are not deposited in the instrumentation directory because the instrumentation directory may be shared by other tests. The results directory is different when you run the test with or without the `-keep` option.

  When you run your test without the `-keep` option the results directory contains the following files:

  – `COV_DESC`: description file of experiment.

  – `COUNTS_<`*exe*`>`: dounts file for each executable; <*exe*> is an executable file name.

– `USER_SELECTIONS`: instrumentation criteria.

When you run your test with the `-keep` option the results directory contains the following files:

– `COV_DESC`: description file of experiment.

– `COUNTS_ <exe>`: counts file for each executable; *exe* is an executable file name.

– `USER_SELECTIONS`: instrumentation criteria.

– `COUNTS_<n>`: basic block and branch counts database.

Usage Model

This section describes the steps in testing:

- "Single Test Analysis Process", page 5, shows the general steps in conducting a test.
- "Automated Testing", page 11, discusses using scripts to automate your testing.
- "Additional Coverage Testing", page 13, describes strategies using multiple tests.

Single Test Analysis Process

In performing coverage analysis for a single test, you typically go through the following steps:

1. **Plan your test**. Test tools are only as good as the quality and completeness of the tests themselves.
2. **Create (or reuse) an instrumentation file.** The instrumentation file defines the coverage data you wish to collect in this test. You can define:
   - COUNTS: three types of count items perform tracking.
     – `bbcounts` tracks execution of basic blocks.
     – `fpcounts` counts calls to functions through function pointers.
     – `branchcounts` tracks branches at the assembly language level.
   - INCLUDE/EXCLUDE: lets you define a subset of functions to be covered. INCLUDE adds the named functions to the current set of functions.

EXCLUDE removes the named functions from the set of functions. Simple pattern matching is supported for pathnames and function names. The basic component for inclusion/exclusion is of the form:

<*shared library* | *program name*>:<*functionlist*>

INCLUDE, EXCLUDE, and CONSTRAIN (see below) play a major role in working with DSOs. Tester instruments all DSOs in an executable whether you are testing them or not, so it is necessary to restrict your coverage accordingly. By default, the directory `/usr/tmp/cvinstrlib/CacheExclude` is used as the excluded DSOs cache and `/usr/tmp/cvinstrlib/CacheInclude` as the included DSOs cache. If you wish to override these defaults, set the CVINSTRLIB environment variable to the desired cache directory.

- CONSTRAIN: equivalent to EXCLUDE *, INCLUDE <*subset*>. Thus, the only functions in the test will be those named in the CONSTRAIN subset. You can constrain the set of functions in the program to either a list of functions or a file containing the functions to be constrained. The function list file format is:

  ```
  function_1
  function_2
  function_3
  ...
  ```

  You can use the `-file` option to include an ASCII file containing all the functions as follows:

  `CONSTRAIN -file` *filename*

The default instrumentation file `/usr/WorkShop/usr/lib/WorkShop/Tester/default_instr_file` contains:

```
#
# Coverage instrumentation normally consists of tracing
# basic block execution, function pointer calls, and branches
#

COUNTS -bbcounts -fpcounts -branchcounts



#
# Exclude instrumentation of any DSOs found under the system
```

```
# library directories (including both system libraries and the
# runtime linker 'rld')
#

EXCLUDE /lib/* : *
EXCLUDE /lib32/* : *
EXCLUDE /lib64/* : *

EXCLUDE /usr/lib/* : *
EXCLUDE /usr/lib32/* : *
EXCLUDE /usr/lib64/* : *



#
# Exclude instrumentation of the C++ "std" namespace
#

EXCLUDE * : std::*



#
# Exclude instrumentation of compiler and implementation specific
# functions that start with the underscore character
#

EXCLUDE * : _*
```

The excluded items are all dynamically shared objects that might interfere with the testing of your main program.

Note: If you do not use the `default_instr_file` file, functions in shared libraries will be included by default, unless your instrumentation file excludes them.

The minimum instrumentation file contains the line:

```
COUNTS -bbcounts
```

You create an instrumentation file using your preferred text editor. Comments are allowed only at the beginning of a new line and are designated by the "#" character. Lines can be continued using a back slash (\) for lists separated with commas. White space is ignored. Keywords are case insensitive. Options and user-supplied names are case sensitive. All lines are additive to the overall experiment description.

Here is a partial instrument file:

```
COUNTS -bbcounts -fpcounts -branchcounts
# defines the counting options, in this case,<
# basic blocks, function pointers, and branches.
CONSTRAIN program:abc, xdr*, functionF, \
classX::methodY, *::methodM, functionG
# constrains the set of functions in the
# ‘‘program’’ to the list of user specified functions
EXCLUDE libc.so.1:*
...
```

Note: Instrumentation can increase the size of a program two to five times. Using DSO caching and sharing can alleviate this problem.

3. **Apply the instrument file to the target executable(s).**

   This is the instrumentation process. You can specify a single executable or more than one if you are creating other processes through `fork`, `exec`, or `sproc`.

   The command line interface command is `runinstr`. The graphical user interface equivalent is the **Run Instrumentation** selection in the **Test** menu.

   The effect of performing a run instrument operation is shown in Figure 1-1. An instrumentation directory is created (`.../ver##<n>`). It contains the instrumented executable and other files used in instrumentation.

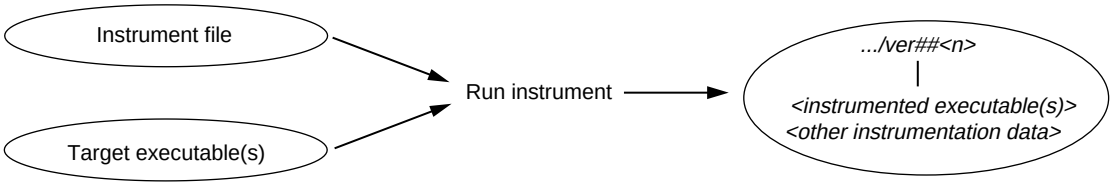

Figure 1-1 Instrumentation Process

4. **Create the test directory.** This part of the process creates a test data directory (`test0000`) containing a test description file named `TDF`. See Figure 1-2, page 9.

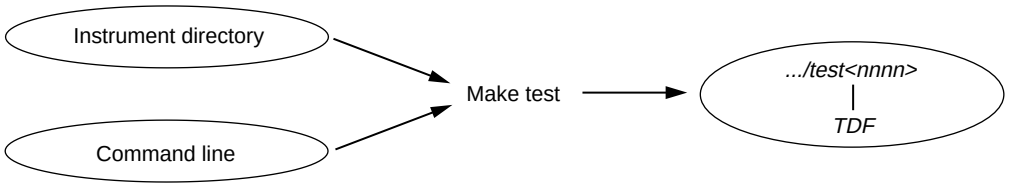

Figure 1-2 Make Test Process

Tester names the test directory `test0000` by default and increments it automatically for subsequent make test operations. You can supply your own name for the test directory if you prefer.

The TDF file contains information necessary for running the test. A typical `TDF` file contains the test name, type, instrument directory, description, and list of executables. In addition, for a test set or test group, the TDF file contains a list of subtests.

Note that the Instrument Directory can be either the instrumentation directory itself (such as `ver##0`) or a directory containing one or more instrumentation subdirectories.

The command line interface command is `mktest`. The graphical user interface equivalent is the **Make Test** selection in the **Test** menu.

5. **Run the instrumented version of the executable to collect the coverage data.** This creates a subdirectory (`exp##0`) under the test directory in which results from the current experiment will be placed.

   See Figure 1-3, page 9. The commands to run a test use the most recent instrumentation directory version unless you specify a different directory.

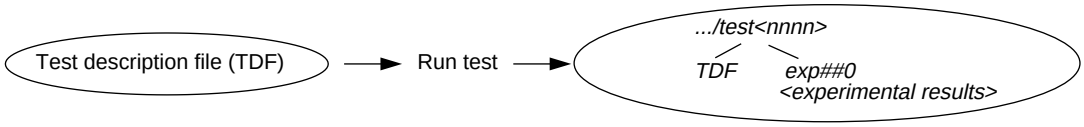

Figure 1-3 Run Test Process

The command-line interface command is `runtest`. The graphical user interface equivalent is the **Run Test** selection in the **Test** menu.

6. **Analyze the results.** Tester provides a variety of column-based presentations for analyzing the results. The data can be sorted by a number of criteria. In addition, the graphical user interface can display a call graph indicating coverage by function and call.

   The Tester interface provides many kinds of queries for performing analysis on a single test. Table 1-1, page 10, shows query commands for a single test that are available either from the command line or the graphical user interface Queries menu.

Table 1-1 Common Queries for a Single Test

Command Line	Graphical User Interface	Description
`lsarc`	**List Arcs**	Shows the function arc coverage. An *arc* is a call from one function to another.
`lsblock`	**List Blocks**	Shows basic block count information.
`lsbranch`	**List Branches**	Shows the count information for assembly language branches.
`lsfun`	**List Functions**	Shows coverage by function.
`lssum`	**List Summary**	Provides a summary of overall coverage.
`lsline`	**List Line Coverage**	Shows coverage for native source lines.
`cattest`	**Describe Test**	Describes the test details.
`diff`	**Compare Test**	Shows the difference in coverage between programs.
`lsinstr`	**List Instrumentation**	Show instrumentation details for a test.

Other queries are accessed differently from either interface.

- `lscall`: shows a function graph indicating caller and callee functions and their counts. From the graphical user interface, function graphs are accessed from a **Call Tree View** (**Views** menu selection).

- `lssource`: displays the source or assembly code annotated with the execution count by line. From the graphical user interface, you access source or assembly code from a **Source View** (using the **Source** button) or a **Disassembly View** (using the **Disassembly** button), respectively.

The queries available in the graphical user interface are shown in Figure 1-4, page 11.

Figure 1-4 The **Queries** Menu from the Main Tester Window

Automated Testing

Tester is best suited to automated testing of command-line programs, where the test behavior can be completely specified at the invocation. Command-line programs let you incorporate contextual information, such as environment variables and current working directory.

Automated testing of server processes in a client-server application proceeds basically the same as single-program cases except that startup time introduces a new factor. Tester can substantially increase the startup time of your target process so that the instrumented target process will run somewhat slower than the standard, uninstrumented one. Tests which start a server, wait a while for it to be ready, and then start the client will have to wait considerably longer. The additional time depends on the size and complexity of the server process itself and on how much and what kind of data you have asked Tester to collect. You will have to experiment to see how long to wait.

Automated testing of interactive or nondeterministic tests is somewhat harder. These tests are not completely determined by their command line; they can produce different results (and display different coverage) from the same command line, depending upon other factors, such as user input or the timing of events. For tests such as these, Tester provides a `-sum` argument to the runtest command. Normally each test run is treated as an independent event, but when you use `runtest -sum`, the coverage from each run is added to the coverage from previous runs of the same test case. Other details of the coverage measurement process are identical to the first case.

In each case, you first need to instrument your target program, then run the test, sum the test results if desired, and finally analyze the results. There are two general approaches to applying `cvcov` in automated testing

- If you have not yet created any test scripts or have a small number of tests, you should create a script that makes each test individually and then runs the complete test set. Example 1-1 shows a script that automates a test program called `target` with different arguments:

  Example 1-1 Making Tests and Running Them

  ```
  # instrument program
  cvcov runinstr -instr_file instrfile mypath/target
  # test machinery
  # make all tests
  cvcov mktest -cmd ‘‘target A B C’’ -testname test0001
  cvcov mktest -cmd ‘‘target D E F’’ -testname test0002
  ...
  # define testset to include all tests
  cvcov lstest > mytest_list
  cvcov mktset -list mytest_list -testname mytestset
  # run all tests in testset and sum up results
  cvcov runtest mytestset
  ```

- If you have existing test scripts of substantial size or an automated test machinery setup, then you may find it straightforward to embed Tester by replacing each test line with a script containing two Tester command lines for making and running the test and then accumulating the results in a testset, such as in Example 1-2. Of course, you can also rewrite the whole test machinery as described in Example 1-1, page 12.

Example 1-2 Applying a Make-and-Run Script

```
# instrument program
cvcov runinstr -instr_file instrfile mypath/target
# test machinery
# make and run all tests
make_and_run ''target A B C''
make_and_run ''target D E F''
...
# make testset
cvcov lstest > mytestlist
cvcov mktset -list mytestlist -testname mytestset
# accumulate results
cvcov runtest mytestset
```

where the `make_and_run` script is:

```
#!/bin/sh
testname='cvcov mktest -instr_dir /usr/tmp -cmd ''$*''''
testname='expr ''$testname'' : ''.*Made test directory: '.*''''
cvcov runtest $testname
```

Note that both examples use simple testset structures—these could have been nested hierarchically if desired.

After running your test machinery, you can use `cvcov` or `cvxcov` to analyze your results. Make sure that your test machinery does not remove the products of the test run (even if the test succeeds), or it may destroy the test coverage data.

Additional Coverage Testing

After you have created and run your first test, you typically need additional testing. Here are some scenarios.

- You can define a test set so that you can vary your coverage using the same instrumentation. You can analyze the new tests singly or you can combine them in a set and look at the cumulative results. If the tests are based on the same executable, they can share the same instrumentation file. You can also have a test set with tests based on different executables but they should have the same instrumentation file.
- You can change the instrumentation criteria to gather different counts or examine a different set of functions.

- You can create a script to run tests in batch mode (command line interface only).
- You can run different programs that use a common dynamically shared object (DSO) and accumulate test coverage for a test group containing the DSO.
- You can run the same tests using the same instrumentation criteria for two versions of the same program and compare the coverage differences.
- You can run a test multiple times and sum the result over the runs. This is typically used for GUI-based applications.

As you conduct more tests, you will be creating more directories. A typical coverage testing hierarchy is shown in Figure 1-5.

There are two different instrumentation directories, `ver##0` and `ver##1`. The test directory `test0000` contains results for a single experiment that uses the instrumentation from `ver##0`. The number in the name of the experiment results directory corresponds to the number of the instrumentation directory. Test directory `test0001` has results for two experiments corresponding to both instrumentation directories, `ver##0` and `ver##1`.

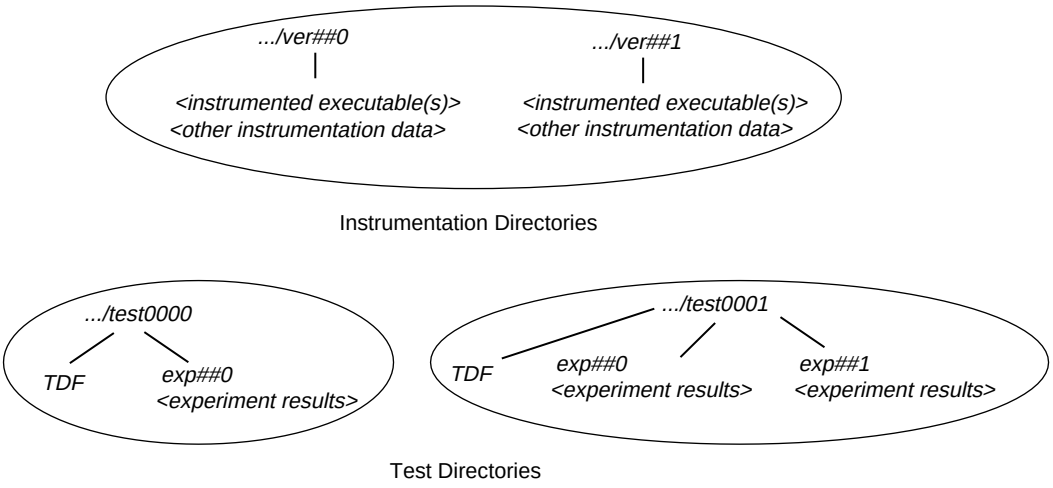

Figure 1-5 Typical Coverage Testing Hierarchy

Tester Command Line Interface Tutorial

The tutorials in this chapter are based on simple programs written in C. To run them, you need the C compiler. The chapter has the following sections:

- "Setting Up the Tutorials", page 15, shows you how to run the script that creates the files needed for the tutorials.
- "Tutorial #1: Analyzing a Single Test", page 16, takes you through the steps of performing coverage analysis for a single test.
- "Tutorial #2: Analyzing a Test Set", page 21, discusses creating additional tests to achieve full coverage.
- "Tutorial #3: Optimizing a Test Set", page 27, explains how to fine-tune a test set to eliminate redundant tests.

If you are going to run these tutorials, you must run them in order; each tutorial builds on the results of previous tutorials.

If at any time a command syntax is not clear, enter the following:

`cvcov help` *commandname*

Setting Up the Tutorials

1. Enter the following commands to set up the tutorials:

   ```
   % cp -r /usr/demos/WorkShop/Tester /usr/tmp/tutorial
   % cd /usr/tmp/tutorial
   % echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > alphabet
   % make -f Makefile.tutorial copyn
   ```

   This moves some scripts and source files used in the tutorial to `/usr/tmp/tutorial`, creates a test file named `alphabet`, and makes a simple program, `copyn`, which copies *n* bytes from a source file to a target file.

2. To see how the program works, try a simple test by typing:

   ```
   % copyn alphabet targetfile 10
   % cat targetfile
   ABCDEFGHIJ
   ```

You should see the first 10 bytes of `alphabet` copied to `targetfile`.

Tutorial #1: Analyzing a Single Test

Tutorial #1 discusses the following topics:

Instrumenting an Executable

This is the first step in providing test coverage. The user defines the instrumentation criteria in an instrumentation file.

1. Enter the following to see the instrumentation directives in the file `tut_instr_file` used in the tutorials:

```
% cat tut_instr_file
COUNTS -bbcounts -fpcounts -branchcounts
CONSTRAIN main, copy_file
```

   We will be getting all counting information (blocks, functions, branches, and arcs) for the two functions specified in the CONSTRAIN directive, `main` and `copy_file`.

2. Enter the following command to instrument `copyn`:

```
% cvcov runinstr -instr_file tut_instr_file copyn
        /lib32/rld
        /usr/lib32/libssrt.so
        /usr/lib32/libss.so
        /usr/lib32/libc.so.1
cvcov: Instrument "copyn" of version "0" succeeded.
```

   Directory `ver##0` has been created by default. This contains the instrumented executable, `copyn.pixie`, and other instrumentation data.

Making a Test

A *test* defines the program and arguments to be run, instrument directory, executables, and descriptive information about the test.

1. Enter the following to make a test:

   ```
   % cvcov mktest -cmd "copyn alphabet targetfile 20"
   ```

   You will see the following message:

   ```
   cvcov: Made test directory: "/var/tmp/tutorial/test0000"
   ```

   Directory `test0000` has been created by default. It contains a single file, `TDF`, the test description file.

   Note: The directory `/var/tmp` is linked to `/usr/tmp`.

2. Enter the following to get a textual listing of the test:

   ```
   % cvcov cattest test0000
   Test Info              Settings
   ------------------------------------------------------
   Test                   /var/tmp/tutorial/test0000
   Type                   single
   Description
   Command Line           copyn alphabet targetfile 20
   Number of Exes         1
   Exe List               copyn
   Instrument Directory   /var/tmp/tutorial/
   Experiment List
   ```

Running a Test

To run a test, we use technology from the WorkShop Performance Analyzer. The instrumented process is set to run, and a monitor process (`cvmon`) captures test coverage data by interacting with the WorkShop process control server (`cvpcs`).

1. Enter the following command:

   ```
   % cvcov runtest test0000
   ```

2. You will see the following message:

```
cvcov: Running test "/var/tmp/tutorial/test0000" ...
/var/tmp/tutorial//ver##0/copyn.pixie alphabet targetfile 20
```

Now the directory `test0000` contains the directory `exp##0`, which contains the results of the first test experiment.

Analyzing Test Coverage Data

You can analyze test coverage data many ways. In this tutorial, we will illustrate a simple top-down approach. We will start at the top to get a summary of overall coverage, proceed to the function level, and go finally to the actual source lines.

1. Enter the following to get the summary:

   ```
   % cvcov lssum test0000
   ```

   You will see the display shown in Example 2-1.

   Example 2-1 lssum Example

```
% cvcov lssum test0000

% cvcov lssum test0000
Coverages          Covered     Total       % Coverage    Weight
-----------------------------------------------------------------
Function           3           3           100.00%       0.400
Source Line        22          33          66.67%        0.200
Branch             0           10          0.00%         0.200
Arc                8           18          44.44%        0.200
Block              24          49          48.98%        0.000
Weighted Sum                               62.22%        1.000
```

   Although both functions have been covered, there is incomplete coverage for source lines, branches, arcs, and blocks.

   Note: Items are highlighted on your screen to emphasize null coverage. As a convention in this manual, highlighting or user input is in boldface.

2. Enter the following to look at the line count information for the `main` function:

   ```
   % cvcov lssource main test0000
   ```

This produces a source listing annotated with counts, shown in Example 2-2.

Example 2-2 lssource Example

```
% cvcov lssource main test0000
Counts  Source
-------------------------------------------------------------------
        #include <stdio.h>
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>

        #define OPEN_ERR         1
        #define NOT_ENOUGH_BYTES 2
        #define SIZE_0           3

        int copy_file();

        main (int argc, char *argv[])
1       {
            int bytes, status;

1           if( argc < 4){
0               printf("copyn: Insufficient arguments.\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
            }
1           if( argc > 4 ) {
0               printf("Error: Too many arguments\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
            }
1           bytes = atoi(argv[3]);
1           if(( status = copy_file(argv[1], argv[2], bytes)) >0){
0              switch ( status) {
                   case SIZE_0:
0                      printf("Nothing to copy\n");
0                      break;
                   case NOT_ENOUGH_BYTES:
0                      printf("Not enough bytes\n");
0                      break;
                   case OPEN_ERR:
```

```
0                       printf("File open error\n");
0                       break;
                     }
0                    exit(1);
                  }
1              }

               int copy_file( source, destn, size)
               char *source, *destn;
               int size;
1              {
                   char *buf;
                   int fd1, fd2;
                   struct stat fstat;
1                  if( (fd1 = open( source, O_RDONLY)) <= 0){
0                      return OPEN_ERR;
                   }
1                  stat( source, &fstat);
1                  if( size <= 0){
0                      return SIZE_0;
                   }
1                  if( fstat.st_size < size){
0                      return NOT_ENOUGH_BYTES;
                   }
1                  if( (fd2 = creat( destn, 00777)) <= 0){
0                      return OPEN_ERR;
                   }
1                  buf = (char *)malloc(size);

1                 read( fd1, buf, size);
1                 write( fd2, buf, size);
1                 return 0;
               }
```

Notice that the 0-counted lines appear in a highlight color. In this example, the lines with 0 counts occur where there is an error condition. This is our first good look at branch and block coverage at the source line level. The branch and block coverage in the summary are at the assembly language level.

Tutorial #2: Analyzing a Test Set

In the second tutorial, we are going to create additional tests with the objective of achieving 100% overall coverage. From examining the source code in Example 2-2, page 19, it seems that the 0-count lines in `main` and `copy_file` are due to error-checking code that is not tested by `test0000`.

Note: This tutorial needs `test0000`, which was created in the previous tutorial.

The script `tut_make_testset` is supplied to demonstrate how to set up this test set.

1. Enter **`sh -x tut_make_testset`** to run the script.

   Example 2-3 shows the first portion of the script (as it runs), in which the individual tests are created. The `tut_make_testset` script uses `mktest` to create eight additional tests. The tests `test0001` and `test0002` pass too few and too many arguments, respectively. `test0003` attempts to copy from a nonexistent file named `no_file`. `test0004` attempts to pass 0 bytes, which is illegal. `test0005` attempts to copy 20 bytes from a file called `not_enough`, which contains only one byte. In `test0006`, we attempt to write to a directory without proper permission. `test0007` tries to copy too many bytes. In `test0008`, we attempt to copy from a file without read permission.

   Example 2-3 `tut_make_testset` Script: Making Individual Tests

```
% sh -x tut_make_testset
+ cvcov mktest -cmd copyn alphabet target -des not enough arguments
cvcov: Made test directory: "/var/tmp/tutorial/test0001"

+ cvcov mktest -cmd copyn alphabet target 20 extra_arg \
-des too many arguments
cvcov: Made test directory: "/var/tmp/tutorial/test0002"

+ cvcov mktest -cmd copyn no_file target 20 -des cannot access file
cvcov: Made test directory: "/var/tmp/tutorial/test0003"

+ cvcov mktest -cmd copyn alphabet target 0 -des pass bad size arg
cvcov: Made test directory: "/var/tmp/tutorial/test0004"

+ echo a

+ 1> not_enough
```

```
+ cvcov mktest -cmd copyn not_enough target 20 -des not enough data \
(less bytes than requested) in original file
cvcov: Made test directory: "/var/tmp/tutorial/test0005"

+ cvcov mktest -cmd copyn alphabet /usr/bin/target 20 \
-des cannot create target executable due to permission problems
cvcov: Made test directory: "/var/tmp/tutorial/test0006"

+ ls -ld /usr/bin
drwxr-xr-x    3 root     sys         3584 May 12 18:25 /usr/bin

+ cvcov mktest -cmd copyn alphabet targetfile 200
-des size arg too big
cvcov: Made test directory: "/var/tmp/tutorial/test0007"

+ cvcov mktest -cmd copyn /usr/adm/sulog targetfile 20 \
-des no read permission on source file
cvcov: Made test directory: "/var/tmp/tutorial/test0008"
```

After the individual tests are created, the script uses `mktset` to make a new test set and `addtest` to include the new tests in the set. Example 2-4 shows the portion of the script in which the test set is created and the individual tests are added to the test set.

Example 2-4 `tut_make_testset` Script: Making and Adding to the Test Set

```
+ cvcov mktset -des full coverage testset -testname tut_testset
cvcov: Made test directory: "/var/tmp/tutorial/tut_testset"

+ cvcov addtest test0000 tut_testset
cvcov: Added "/var/tmp/tutorial/test0000" to "tut_testset"

+ cvcov addtest test0001 tut_testset
cvcov: Added "/var/tmp/tutorial/test0001" to "tut_testset"

+ cvcov addtest test0002 tut_testset
cvcov: Added "/var/tmp/tutorial/test0002" to "tut_testset"

+ cvcov addtest test0003 tut_testset
cvcov: Added "/var/tmp/tutorial/test0003" to "tut_testset"
```

```
+ cvcov addtest test0004 tut_testset
cvcov: Added "/var/tmp/tutorial/test0004" to "tut_testset"

+ cvcov addtest test0005 tut_testset
cvcov: Added "/var/tmp/tutorial/test0005" to "tut_testset"

+ cvcov addtest test0006 tut_testset
cvcov: Added "/var/tmp/tutorial/test0006" to "tut_testset"

+ cvcov addtest test0007 tut_testset
cvcov: Added "/var/tmp/tutorial/test0007" to "tut_testset"

+ cvcov addtest test0008 tut_testset
cvcov: Added "/var/tmp/tutorial/test0008" to "tut_testset"
```

2. Enter **`cvcov cattest tut_testset`** to check that the new test set was created correctly.

   This is shown in Example 2-5. The index numbers in brackets in the subtest list are used to identify the individual tests as part of a test set. This index is used to list the contribution of each test.

   Example 2-5 Contents of the New Test Set

```
% cvcov cattest tut_testset
Test Info              Settings
-------------------------------------------------------
Test                   /var/tmp/tutorial/tut_testset
Type                   set
Description            full coverage testset
Number of Exes         1
Exe List               copyn
Number of Subtests     9
Subtest List
                       [0] /var/tmp/tutorial/test0000
                       [1] /var/tmp/tutorial/test0001
                       [2] /var/tmp/tutorial/test0002
                       [3] /var/tmp/tutorial/test0003
                       [4] /var/tmp/tutorial/test0004
                       [5] /var/tmp/tutorial/test0005
                       [6] /var/tmp/tutorial/test0006
                       [7] /var/tmp/tutorial/test0007
```

```
                                [8] /var/tmp/tutorial/test0008
Experiment List
```

3. Enter the following to run the tests in the test set:

   ```
   % cvcov runtest tut_testset
   ```

   By applying the `runtest` command to the test set, we can run all the tests together. See Example 2-6. When you run a test set, only tests without results are run; tests that already have results will not be run again. In this case, `test0000` has already been run. If you need to rerun a test, you can do so using the `-force` flag.

   Example 2-6 Running the New Test Set

```
% cvcov runtest tut_testset
cvcov: Running test "/var/tmp/tutorial/test0000" ...
cvcov: Running test "/var/tmp/tutorial/test0001" ...
/var/tmp/tutorial//ver##0/copyn.pixie alphabet target
copyn: Insufficient arguments.
Usage: copyn f1 f2 bytes
cvcov: Running test "/var/tmp/tutorial/test0002" ...
/var/tmp/tutorial//ver##0/copyn.pixie alphabet target 20 extra_arg
Error: Too many arguments
Usage: copyn f1 f2 bytes
cvcov: Running test "/var/tmp/tutorial/test0003" ...
/var/tmp/tutorial//ver##0/copyn.pixie no_file target 20
File open error
cvcov: Running test "/var/tmp/tutorial/test0004" ...
/var/tmp/tutorial//ver##0/copyn.pixie alphabet target 0
Nothing to copy
cvcov: Running test "/var/tmp/tutorial/test0005" ...
/var/tmp/tutorial//ver##0/copyn.pixie not_enough target 20
Not enough bytes
cvcov: Running test "/var/tmp/tutorial/test0006" ...
/var/tmp/tutorial//ver##0/copyn.pixie alphabet /usr/bin/target 20
File open error
cvcov: Running test "/var/tmp/tutorial/test0007" ...
/var/tmp/tutorial//ver##0/copyn.pixie alphabet targetfile  200
Not enough bytes
cvcov: Running test "/var/tmp/tutorial/test0008" ...
/var/tmp/tutorial//ver##0/copyn.pixie /usr/adm/sulog targetfile 20
File open error
```

4. Enter **`cvcov lssum tut_testset`** to list the summary for the test set.

   Example 2-7 shows the results of the tests in the new test set with `lssum`.

   Example 2-7 Examining the Results of the New Test Set

```
% cvcov lssum tut_testsetCoverages          Covered     Total       % Coverage    Weight
-----------------------------------------------------------------------
Function           3           3           100.00%       0.400
Source Line        33          33          100.00%       0.200
Branch             9           10          90.00%        0.200
Arc                18          18          100.00%       0.200
Block              46          49          93.88%        0.000
Weighted Sum                               98.00%        1.000
```

   Note: Block (basic block) weight will always be different based depending on compile options and compiler versions.

5. Enter **`cvcov lssource main tut_testset`** to see the coverage for the individual source lines as shown in Example 2-8, page 25.

   Example 2-8 Source with Counts

```
% cvcov lssource main tut_testset
Counts  Source
-----------------------------------------------------------------------
        #include <stdio.h>
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>

        #define OPEN_ERR         1
        #define NOT_ENOUGH_BYTES 2
        #define SIZE_0           3

        int copy_file();

        main (int argc, char *argv[])
9       {
            int bytes, status;
```

```
9           if( argc < 4){
1               printf("copyn: Insufficient arguments.\n");
1               printf("Usage: copyn f1 f2 bytes\n");
1               exit(1);
            }
8           if( argc > 4 ) {
1               printf("Error: Too many arguments\n");
1               printf("Usage: copyn f1 f2 bytes\n");
1               exit(1);
            }
7           bytes = atoi(argv[3]);
7           if(( status = copy_file(argv[1], argv[2], bytes)) >0){
6              switch ( status) {
                   case SIZE_0:
1                      printf("Nothing to copy\n");
1                      break;
                   case NOT_ENOUGH_BYTES:
2                      printf("Not enough bytes\n");
2                      break;
                   case OPEN_ERR:
3                      printf("File open error\n");
3                      break;
               }
6              exit(1);
            }
1       }

        int copy_file( source, destn, size)
        char *source, *destn;
        int size;
7       {
            char *buf;
            int fd1, fd2;
            struct stat fstat;
7           if( (fd1 = open( source, O_RDONLY)) <= 0){
2               return OPEN_ERR;
            }
5           stat( source, &fstat);
5           if( size <= 0){
1               return SIZE_0;
            }
```

```
4           if( fstat.st_size < size){
2               return NOT_ENOUGH_BYTES;
            }
2           if( (fd2 = creat( destn, 00777)) <= 0){
1               return OPEN_ERR;
            }
1           buf = (char *)malloc(size);

1          read( fd1, buf, size);
1          write( fd2, buf, size);
7          return 0;
        }
```

As you look at the source code, notice that all lines are covered.

6. Enter **`cvcov lssource -asm main tut_testset`** to see the coverage for the individual assembly lines.

   When we list the assembly code using `lssource -asm`, we find that not all blocks and branches are covered at the assembly level. This is due to compilation with the `-g` flag, which adds debugging code that can never be executed.

   Enter **`cvcov lsline tut_testset`** to see the coverage at the source line level. Notice that 100% of the lines have been covered.

Tutorial #3: Optimizing a Test Set

Tester lets you look at the individual test coverages in a test set. When you put together a set of tests, you may want to improve the efficiency of your coverage by eliminating redundant tests. The `lsfun`, `lsblock`, and `lsarc` commands all have the `-contrib` option, which displays coverage result contributions by individual tests. We will now look at the contributions by tests for the test set we just ran, `tut_testset`.

Note: This tutorial needs `tut_testset` and all its subtests; these were created in the previous tutorial.

---

1. Enter **`cvcov lsfun -contrib -pretty tut_testset`** to see the function coverage test contribution.

Example 2-9, page 28, shows how the test set covers functions. Note that the subtests are identified by index numbers; use `cattest` if you need to map these results back to the test directories.

Example 2-9 Test Contributions by Function

```
% cvcov lsfun -contrib -pretty tut_testset
Functions    Files            Counts
-----------------------------------------
main         copyn.c          9
copy_file    copyn.c          7
main         rld_startup.c    1

Functions    Files            [0]    [1]    [2]    [3]    [4]    [5]
-------------------------------------------------------------------------
main         copyn.c          1      1      1      1      1      1
copy_file    copyn.c          1      0      0      1      1      1
main         rld_startup.c    1      0      0      0      0      0

Functions    Files            [6]    [7]    [8]
-------------------------------------------------
main         copyn.c          1      1      1
copy_file    copyn.c          1      1      1
main         rld_startup.c    0      0      0
```

At the function level, each test covers both functions except for Tests [1] and [2]. The information here is not sufficient to tell us if we have optimized the test set. To do this, we must look at contributions at the arc and block levels. Tester shows arc and block coverage information by test when you apply the `-contrib` flag to `lsarc` and `lsblock`, respectively.

2. Enter **`cvcov lsarc -contrib -pretty tut_testset`** to see the arc coverage test contribution.

   Example 2-10, page 29, shows the individual test contributions. Notice that Tests [5] and [7] have identical coverage to each other; so do Tests [3] and [8].

   We can get additional information by looking at block coverage, confirming our hypothesis about redundant tests.

Example 2-10 Arc Coverage Test Contribution Portion of Report

```
% cvcov lsarc -contrib -pretty tut_testset
Callers      Callees      Line        Files       Counts
-------------------------------------------------------------
main         copy_file    27          copyn.c     7
main         printf       17          copyn.c     1
main         printf       18          copyn.c     1
main         __exit       19          copyn.c     1
main         printf       22          copyn.c     1
main         printf       23          copyn.c     1
main         __exit       24          copyn.c     1
main         atoi         26          copyn.c     7
main         printf       30          copyn.c     1
main         printf       33          copyn.c     2
main         printf       36          copyn.c     3
main         __exit       39          copyn.c     6
copy_file    _open        50          copyn.c     7
copy_file    _stat        53          copyn.c     5
copy_file    _creat       60          copyn.c     2
copy_file    malloc       63          copyn.c     1
copy_file    _read        65          copyn.c     1
copy_file    _write       66          copyn.c     1

Callers      Callees      Line        Files       [0]    [1]    [2]    [3]   [4]    [5]
-------------------------------------------------------------------------------------
main         copy_file    27          copyn.c     1      0      0      1     1      1
main         printf       17          copyn.c     0      1      0      0     0      0
main         printf       18          copyn.c     0      1      0      0     0      0
main         __exit       19          copyn.c     0      1      0      0     0      0
main         printf       22          copyn.c     0      0      1      0     0      0
main         printf       23          copyn.c     0      0      1      0     0      0
main         __exit       24          copyn.c     0      0      1      0     0      0
main         atoi         26          copyn.c     1      0      0      1     1      1
main         printf       30          copyn.c     0      0      0      0     1      0
main         printf       33          copyn.c     0      0      0      0     0      1
main         printf       36          copyn.c     0      0      0      1     0      0
main         __exit       39          copyn.c     0      0      0      1     1      1
copy_file    _open        50          copyn.c     1      0      0      1     1      1
copy_file    _stat        53          copyn.c     1      0      0      0     1      1
copy_file    _creat       60          copyn.c     1      0      0      0     0      0
```

```
copy_file    malloc       63          copyn.c     1      0      0      0      0      0
copy_file    _read        65          copyn.c     1      0      0      0      0      0
copy_file    _write       66          copyn.c     1      0      0      0      0      0

Callers      Callees      Line        Files       [6]    [7]    [8]
-----------------------------------------------------------------------
main         copy_file    27          copyn.c     1      1      1
main         printf       17          copyn.c     0      0      0
main         printf       18          copyn.c     0      0      0
main         __exit       19          copyn.c     0      0      0
main         printf       22          copyn.c     0      0      0
main         printf       23          copyn.c     0      0      0
main         __exit       24          copyn.c     0      0      0
main         atoi         26          copyn.c     1      1      1
main         printf       30          copyn.c     0      0      0
main         printf       33          copyn.c     0      1      0
main         printf       36          copyn.c     1      0      1
main         __exit       39          copyn.c     1      1      1
copy_file    _open        50          copyn.c     1      1      1
copy_file    _stat        53          copyn.c     1      1      0
copy_file    _creat       60          copyn.c     1      0      0
copy_file    malloc       63          copyn.c     0      0      0
copy_file    _read        65          copyn.c     0      0      0
copy_file    _write       66          copyn.c     0      0      0
```

3. Enter the following to see the test contribution to block coverage:

   ```
   % cvcov lsblock -contrib -pretty tut_testset
   ```

   If you examine the results, you will see that Tests [5] and [7] and Tests [3] and [8] are identical.

   Now we can try to tune the test set. If we can remove tests with redundant coverage and still achieve the equivalent overall coverage, then we have tuned our test set successfully. Since the arcs and blocks covered by Test [7] are also covered by Test [5], we can remove either one of them without affecting the overall coverage. The same analysis holds true for Tests [3] and [8].

4. Delete `test0007` and `test0008` as shown in Example 2-11, page 31. Then rerun the test set and look at its summary.

Note that the coverage is retabulated without actually rerunning the tests. The test summary shows that overall coverage is unchanged, thus confirming our hypothesis.

Example 2-11 Test Set Summary after Removing Tests [8] and [7]

```
% cvcov deltest test0008 tut_testset
cvcov: Deleted "/var/tmp/tutorial/test0008" from "tut_testset"

% cvcov deltest test0007 tut_testset
cvcov: Deleted "/var/tmp/tutorial/test0007" from "tut_testset"

% cvcov runtest tut_testset
cvcov: Running test "/var/tmp/tutorial/test0000" ...
cvcov: Running test "/var/tmp/tutorial/test0001" ...
cvcov: Running test "/var/tmp/tutorial/test0002" ...
cvcov: Running test "/var/tmp/tutorial/test0003" ...
cvcov: Running test "/var/tmp/tutorial/test0004" ...
cvcov: Running test "/var/tmp/tutorial/test0005" ...
cvcov: Running test "/var/tmp/tutorial/test0006" ...

% cvcov lssum tut_testset
Coverages                   Covered     Total       % Coverage    Weight
-------------------------------------------------------------------------------
Function                    3           3           100.00%       0.400
Source Line                 33          33          100.00%       0.200
Branch                      9           10          90.00%        0.200
Arc                         18          18          100.00%       0.200
Block                       48          52          92.31%        0.000
Weighted Sum                                        98.00%        1.000
```

Tester Command Line Reference

This chapter describes the `cvcov` commands. It contains the following two subsections:

- "Common `cvcov` Options", page 33, describes the command arguments that are common to more than one command
- "`cvcov` Command Syntax and Description", page 34, describes the specifications with descriptions for each command

A complete description of the `cvcov` commands, including individual arguments, is available on the `cvcov` man page by typing:

```
% man cvcov
```

For examples of `cvcov` usage, see Appendix A, "`cvcov` Command Line Examples", page 105.

Common cvcov Options

This section contains descriptions of some `cvcov` flags and variables that are common to more than one command.

- `[-ver]`: displays the version of `cvcov`. Note that there are no other arguments permitted; you enter: `cvcov -ver`
- `[-v` *versionnumber*`]`: allows you to specify a version of the instrumentation or experiment directory other than the most recent, which is the default.
- `[-contrib]`: shows the list of tests that contributed to coverage for the particular query.
- `[-exe` *exe_name*`]`: lets you specify an executable for coverage testing. This is used when there are multiple executables involved, as in testing processes created by the `fork`, `exec`, or `sproc` command.
- `[-instr_dir` *instr_dir*`]`: allows you to specify an instrumentation directory other than the current working directory, which is the default.
- `[-instr_file` *instr_file*`]`: specifies the instrumentation file, which is an ASCII description of the instrumentation criteria you have selected.

- [-list *list_file*]: specifies a file containing a list of test names to be made part of a test set or group. If no -list option is specified, an empty test set will be created.
- [-r]: (Recursion) lets you specify tests in a hierarchy of subdirectories.
- [-arg]: displays functions with their arguments.
- [-pretty]: displays output aligned in columns. Without -pretty, the output is in columns but more condensed.
- [-sort]: sorts the output by the specified criteria, as follows:
  - function: alphabetically by function
  - diff: by differences in the counting information for coverage type
  - caller: alphabetically by calling function
  - callee: alphabetically by called function
  - count: by counts for current coverage type
  - file: alphabetically by file name
  - type: alphabetically by argument type
- [-functions]: displays list of constrained functions.
- [-pat *func_pattern*]: lets you enter a pattern instead of a complete function name. The pattern can be of the form func_name, dso_:func_name, or 'dso:*'.
- experiment | test_name: lets you specify either the experiment subdirectory or the test directory. The test directory is typically of the form test<*nnnn*>, where <*nnnn*> is a number in a sequence counting from 0000. You can specify your own name. The test directory contains all information about a test including the experiment directory. The experiment directory is typically of the form exp##<*n*>, where <*n*> is a sequential number, counting from 0.

cvcov Command Syntax and Description

This section contains the syntax and description for all cvcov commands in the command line interface. If you need information on command arguments that are not described in this section, please refer back to "Common cvcov Options", page 33.

The most general command is the `help` command, as follows:

`cvcov help` *command_name*

The `help` command prints help on the specified command. If the optional command name is not specified, it prints help for all the commands.

The rest of the commands are divided up into these categories:

- General test commands, described in "General Test Commands", page 35.
- Coverage analysis commands, described in "Coverage Analysis Commands", page 37.
- Test set commands, described in "Test Set Commands", page 39.
- Test group command, described in "Test Group Commands", page 40.

General Test Commands

The following commands support the creation, inspection, modification, and deletion of tests:

- `cvcov cattest [-r]` *test_name*

  Describes the test details for a test, test set, or test group. See Example A-1, page 105, Example A-2, page 105, and Example A-3, page 106.

- `cvcov lsinstr [-exe]` *exe_name* `[-functions] [-v` *versionnumber*`]` *test_name*

  Displays the instrumentation information for a particular test. *exe_name* is the executable targeted for query. The main program is the default if no executable is specified. The `-functions` parameter shows the functions that are included in the coverage experiment. The *versionnumber* parameter allows you to specify the version of the program that was instrumented. You can specify the test directory using the *test_name* parameter. See Example A-4, page 107.

- `cvcov lstest [-r][`*test_name...*`]`

  Lists the test directories in the current working directory. Note that the *test_name* parameter will accept regular expressions for `lstest`.

- `cvcov mktest -cmd` *cmd_line* `[-des` *description*`] [-instr_dir` *directoryname*`] [-testname` *test*`] [`*exe1 exe2* `...]`

Creates a test directory. You specify the program and command line options for the program to be tested. This includes any redirection for `stdin`, `stderr`, or `stdout` as run from the Bourne shell.

The `-cmd` qualifier is mandatory, even if it only includes the program name. If no executables are specified, only the main program is tested. Example A-5, page 107, shows an example of `mktest`, followed by `cattest` to display the contents of the Test Description File (TDF).

- `cvcov rmtest [-r]`*test_name* `...`

  Removes tests and test sets. Note that the *test_name* parameter will accept regular expressions for `rmtest`. It is recommended to separate the test set directory from its test subdirectories and the instrument directory. In this way, `rmtest` will not remove instrumentation data or subtests if you choose to remove the test set only.

- `cvcov runinstr [-instr_dir`*instr_dir*`] [-instr_file` *instr_file*`] [-v` *versionnumber*`]` *executable*

  Adds code to the target executable to enable you to capture coverage data, according to the criteria you specify. The instrument file is an ASCII description of the instrumentation criteria for the experiment. You can also specify the version of the executable and instrument directory.

  You can capture basic block counts, function pointer counts, and branch counts (at the assembly language level). You can use INCLUDE, EXCLUDE, or CONSTRAIN to modify the set of functions covered. CONSTRAIN lets you define a set of functions for the test.

- `cvcov runtest [ -bitcount ][ -compress ][-force] [-keep][-sum] [-v` *versionnumber*`] [-noarc] [-rmsub]` *test_name*

  Runs a test or a set of tests.

  The `-bitcount` flag compresses count data file to be 1-bit-per-count. This option can decrease the database size up to 32 times, although branch count information will be lost.

  The `-compress` flag compresses the experiment database using the standard utility `compress`.

  The `-force` flag forces the test to be run again even if an experiment is present. It uses WorkShop performance tool technology to set up the instrumented process, run the process, and monitor the run, collecting counting information upon exit.

The `-keep` flag retains all performance data collected in the experiment. By default, the performance data is not retained, because it is not required by the coverage tool. The `-sum` flag accumulates (sum over) the coverage data into the existing experiment results. This allows users to run and rerun the same test and accumulate the results in one place.

The `-noarc` flag prevents arc information from being saved in the test database. With the `-noarc` flag, all arc-related queries will not work (for example, `lsarc` and `lscall`).

The `-rmsub` flag removes results for individual subtests for a test set or test group. There will be no data to query if you are querying a subtest. `-noarc` and `-rmsub` save disk space.

Coverage Analysis Commands

After the data has been collected from the test experiments, the user can analyze the data. There are special commands for the various types of coverage available.

- `cvcov lssum [-exe` *exe_name*`] [-weight` *func_factor* `:`*line_factor* `:` *branch_factor* `:` *arc_factor* `:` *block_factor*`]` *experiment* `|` *test_name*

  Shows the overall coverage based on the user-defined weighted average over function, line, block, branch, and arc coverage. See Example A-6, page 108.

- `cvcov lsfun [-arg] [-bf` *filter_type block_filter_value*`] [-blocks] [-branches] [-contrib] [-exe` *exe_name*`] [-ff` *filter_type func_filter_value*`] [-pat` *func_pattern*`] [-pretty] [-rf` *filter_type branch_filter_value*`] [-sort` *count* `|` *file* `|` *function*`]` *experiment* `|`*test_name*

  Lists coverage information for the specified functions in the program that was tested. Several sorting, matching, and filtering techniques are available. For example, you can show the list of functions that have 0 counts (were not covered) in alphabetical order. You can display arguments with the `-arg` flag. See Example A-7, page 108.

- `cvcov lsblock [-addr] [-arg] [-contrib] [-exe` *exe_name*`] [-pat` *func_pattern*`] [-pretty] [-sort` *count*`|`*file*`|`*function*`]` *experiment*`|` *test_name*

  Displays a list of blocks for one or more functions and the count information associated with each block. See Example A-8, page 108.

Blocks are identified by the line numbers in which they occur. If there are multiple blocks in a line, blocks subsequent to the first are shown in order with an index number in parentheses. Be careful before listing all blocks in the program, since this can produce a lot of data. The `-addr` flag show blocks with the PC range instead of the source line number range.

- `cvcov lsbranch [-addr] [-arg] [-exe`*exe_name*`][-pat` *func_pattern*`][-pretty][-sort` *function*`|` *file*`]` *experiment*`|`*test_name*

  Lists coverage information for branches in the program, including the line number at which the branch occurs. See Example A-9, page 109.

  Branch coverage counts assembly language branch instructions that are both taken and not taken. The `-addr` flag show blocks with the PC range instead of the source line number range.

- `cvcov lsarc [-arg] [-callee` *callee_pattern*`] [-caller` *caller_pattern*`] [-contrib][-exe` *exe_name*`] [-pretty][-sort` *caller*`|`*callee*`|` *count*`|`*file*`]` *experiment*`|` *test_name*

  Shows *arc coverage*, that is, the number of arcs taken out of the total possible arcs. See Example A-10, page 110.

  An arc is a function caller-callee pair. Both *callee_pattern* and *caller_pattern* can be specified in the same way as *func_pattern* (used with the `-pat` option) as shown under "Common `cvcov` Options", page 33.

- `cvcov lscall [-arg] [-exe` *exe_name*`][-node` *func_name*`] [-pretty] [-r]` *experiment*`|`*test_name*

  Lists the call graph for the executable with counts for each function. The contribution to this coverage by each test is shown in a separate column. `N/A` means the node is excluded. See Example A-11, page 110.

  A function that has more than one parent and has children is called a *subnode*. Using `-r` will display the subnodes. Subnodes are given their own starting point in the textual call graph. They are identified by a trailing ellipsis (...). For example, see `printf`, `exit`, and `malloc` in Example A-11.

- `cvcov lsline [-arg] [-exe` *exe_name*`] [-pat` *func_pattern*`][-pretty] [-sort` *function*`|`*file*`]` *experiment*`|`*test_name*

  Lists the coverage for native source lines. Use `-arg` to show arguments for functions. If no executable is specified, the main program is the default. Use `-pretty` to provide column-aligned output. See Example A-12, page 110.

- `cvcov lssource [-asm] [-exe` *exe_name*`]` *function experiment test_name*

  Displays the source annotated with line counts. The `-asm` switch displays the assembly level source code annotated with line counts. Lines with 0 counts are highlighted to show the absence of coverage. This is useful for mapping to the source level blocks and branches that were not covered. Lines in functions that were not included in the test appear without count annotations. See Example A-13, page 111.

  Note: `lssource` requires the code to be compiled with the `-g` option.

- `cvcov diff [-arg] [-exe` *exe_name*`] [-functions] [-pretty][-sort diff|function]` *experiment1 experiment2*

  Shows the difference in coverage for different versions of the same program. See Example A-14, page 111.

Test Set Commands

A test set is a named collection of tests and other test sets. Test sets can be hierarchical. For example, `compiler_language_suite` might include `C++_suite`, `C_suite`, and `Fortran_suite`, where `Fortran_suite` is a test set with subdirectories. The following commands support creation, inspection, modification, and deletion of test sets. Both `addtest` and `deltest` are also used with test groups, described in the next section.

- `cvcov mktset [-des` *description*`] [-list` *list_file*`][-testname` *test*`]`

  Makes a test set. If no test name is specified, the command assigns one automatically. See Example A-16, page 112.

- `cvcov addtest` *test_nametest_set_name*`|` *test_group*

  Adds a test or test set to a test set or test group.

- `cvcov deltest` *test_name test_set_name*`|`*test_group*

  Removes a test or test set from a test set or test group.

  Note: Do not use UNIX commands `mv` and `cp` to rename or copy test sets because they are constructed with absolute file paths.

- `cvcov optimize [ -blocks ][ -branches ][ -cbb` *filter_type bb_filter_value* `][ -cbr` *filter_type br_filter_value*`] [ -exe` *exe_name*`] [ -pat` *func_pattern*`] [ -pretty ][ -stat ]`*experiment...*|*test_name* `...`

  Selects the minimum set of tests that give the same coverage or meet the given coverage criteria as the given set.

  The `-blocks` flag shows block coverage for all the selected tests.

  The `-branches` flag shows branch coverage for all the selected tests.

  The `-cbb` *filter_type bb_filter_value* gives the basic block coverage criteria for test selection. The rules are the same as the flag `-bf` of the `lsfun` command.

  The `-cbr` *filter_type br_filter_value* gives the branch coverage criteria for test selection. The rules are the same as the flag `-rf` of `lsfun` command.

  The `-exe` *exe_name* option lets you specify which executable is targeted for test optimization. If no executable is specified, the main program is the default.

  The `-pat` *pattern* option lets you specify DSO patterns for calculation of coverage on test selection. The `-pretty` flag aligns column output.

  The `-stat` flag prints out block and branch coverage for all the selected tests. Without this option, cumulative coverages for block and branch are given.

  The *experiment ...*|*test_name ...* option lets you specify names of experiments or tests to be optimized. Example A-16, page 112, demonstrates how test sets are optimized. In this case, optimizing is applied to all tests matching the expression `test00*`.

Test Group Commands

A *test group* is a collection of programs to be tested that have a common dynamically shared object (DSO). The coverage testing is limited to activity with the DSO so that the arcs and branches that terminate outside of the DSO will not be included. See descriptions of `addtest` and `deltest` in the previous section as well as the following command.

`cvcov mktgroup [-des` *description*`][-list` *list_file*`][-testname` *test*`]` *target1 target2*`...`

This command creates a test group that can contain other tests or test groups. The targets are either the target libraries or DSOs.

Note: Do not use UNIX commands `mv` and `cp` to rename or copy test groups because they are constructed with absolute files paths.

Tester Graphical User Interface Tutorial

This chapter provides a tutorial for the Tester graphical user interface. It covers these topics:

Setting Up the Tutorial

If you have already set up a tutorial directory for the command line interface tutorial, you can continue to use it. If you remove the subdirectories, your directory names will match exactly; if you leave the subdirectories in, you can add new ones as part of this tutorial.

If you would like the test data built automatically, run the following script:

```
/usr/demos/WorkShop/Tester/setup_Tester_demo
```

To set up a tutorial directory from scratch, do the following; otherwise you can skip the rest of this section.

1. Enter the following:

   ```
   % cp -r /usr/demos/WorkShop/Tester /usr/tmp/tutorial
   % cd /usr/tmp/tutorial
   % echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > alphabet
   % make -f Makefile.tutorial copyn
   ```

   This moves some scripts and source files used in the tutorial to `/usr/tmp/tutorial`, creates a test file named `alphabet`, and makes a simple program, `copyn`, which copies *n* bytes from a source file to a target file.

2. To see how the program works, try a simple test by typing the following at the command line:

```
% ./copyn alphabet targetfile 10
% cat targetfile
ABCDEFGHIJ
```

You should see the first 10 bytes of `alphabet` copied to `targetfile`.

Tutorial #1: Analyzing a Single Test

Tutorial #1 discusses the following topics:

- "Invoking the Graphical User Interface", page 44.
- "Instrumenting an Executable", page 47.
- "Making a Test", page 48.
- "Running a Test", page 50.
- "Analyzing the Results", page 51.

These topics are all covered in the following sections.

Invoking the Graphical User Interface

You typically call up the graphical user interface from the directory that will contain your test subdirectories. This section tells you how to invoke the Tester graphical user interface and describes the main window.

Procedure 4-1 Invoking the GUI

1. Enter **`cvxcov`** from the tutorial directory you created previously (for example, `/usr/tmp/tutorial`) to bring up the Tester main window.

   Figure 4-1, page 46, shows the main Tester window with all its menus displayed.

Note: You can also access Tester from the **Admin** menu in other WorkShop tools.

---

2. Observe the features of the Tester window.

   - The **Test Name** field is used to display the current test. You can switch to different tests through this field.

- Test results display in the coverage display area. You display the results by choosing an item from the **Queries** menu. You also can select the format of the data from the **Views** menu.
- The **Source** button lets you bring up the standard **Source View** window with Tester annotations. **Source View** shows the counts for each line included in the test and highlights lines with 0 counts. Lines from excluded functions display but without count annotations.
- The **Disassembly** button brings up the **Disassembly View** window for assembly language source. It operates in a similar fashion to the **Source** button.
- The **Contribution** button displays a separate window with the contributions to the coverage made by each test in a test set or test group.
- A sort button lets you sort the test results by such criteria as function, count, file, type, difference, caller, or callee. The criteria available (shown by the name of the button) depend on the current query.
- The status area displays status messages regarding the test.

  The area below the status area will display special query-specific fields when you make queries.
- You can launch other WorkShop applications from the **Launch Tool** submenu of the **Admin** menu. The applications include the Build Analyzer, Debugger, Parallel Analyzer, Performance Analyzer, and Static Analyzer.

You will also find an icon version of the **Execution View** labeled `cvxcovExec`. It is a shell window for viewing test results as they would appear on the command line.

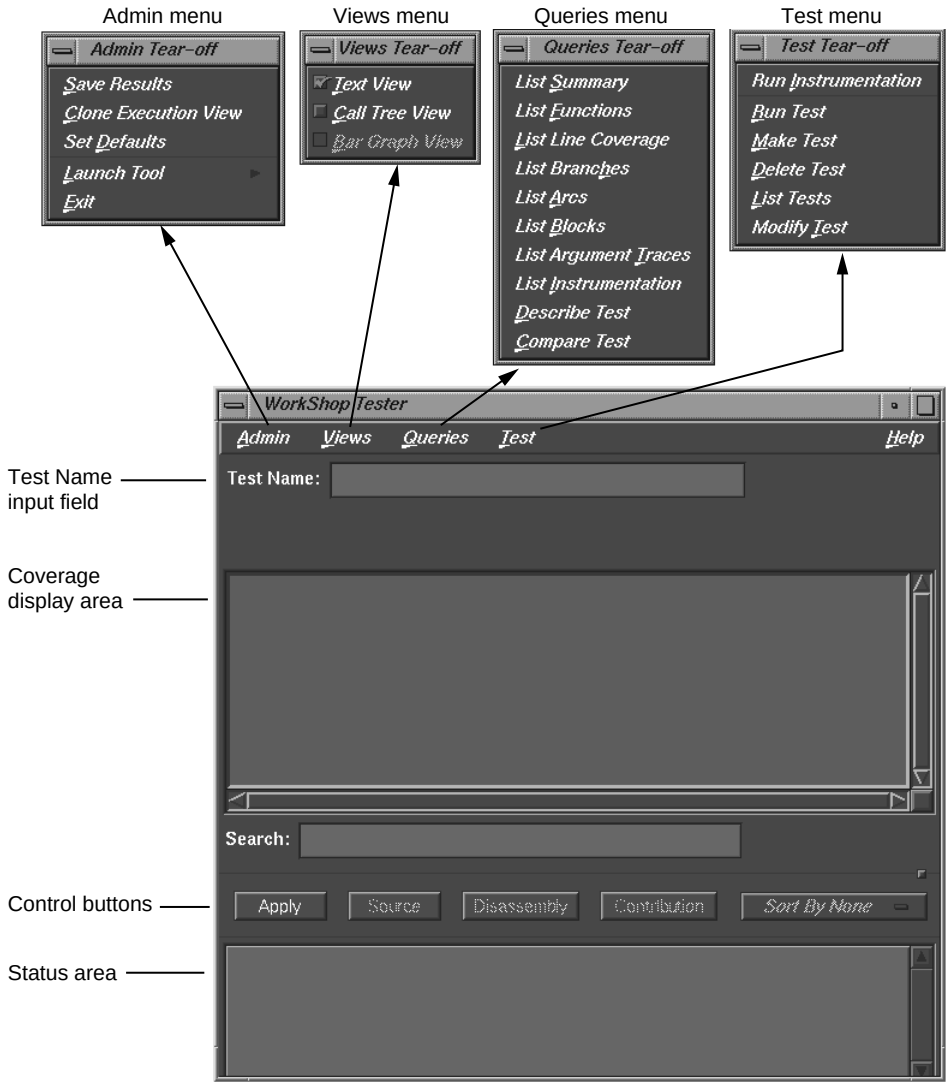

Figure 4-1 Main Tester Window

Instrumenting an Executable

The first step in providing test coverage is to define the instrumentation criteria in an instrumentation file.

Procedure 4-2 Instrumenting an Executable

1. From **Execution View**, enter the following to see the instrumentation directives in the file `tut_instr_file` used in the tutorials:

   ```
   % cat tut_instr_file
   COUNTS -bbcounts -fpcounts -branchcounts
   CONSTRAIN main, copy_file
   TRACE BOUNDS copy_file(size)
   ```

   We will be getting all counting information (blocks, functions, source lines, branches, and arcs) for the two functions specified in the CONSTRAIN directive, `main` and `copy_file`.

2. Select **Run Instrumentation** from the **Test** menu in the Tester main window.

   This process inserts code into the target executable that enables coverage data to be captured. The dialog box shown in Figure 4-2, page 47, displays when **Run Instrumentation** is selected from the **Test** menu.

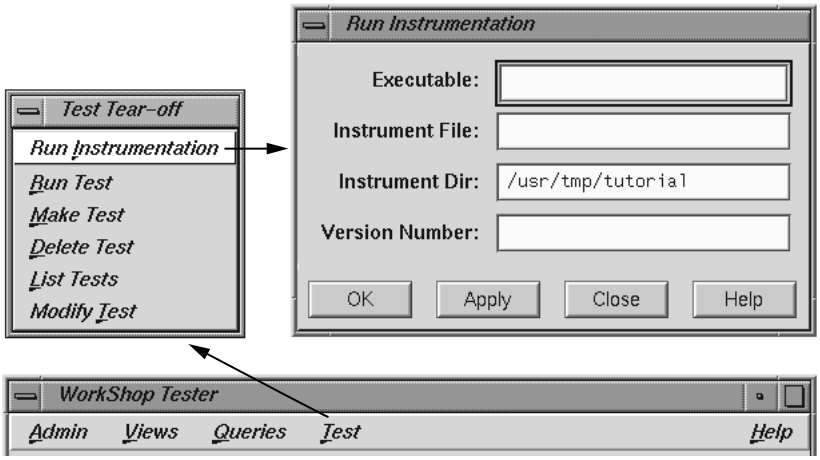

Figure 4-2 Running Instrumentation

3. Enter **copyn** in the **Executable** field.

   The **Executable** field is required, as indicated by the red highlight. You enter the executable in this field.

4. Leave the **Instrument Dir** and **Version Number** fields as is.

   The **Instrument Dir** field indicates the directory in which the instrumented programs are stored. A versioned directory is created (the default is `ver##`*n*, where *n* is 0 the first time and is incremented automatically if you subsequently change the instrumentation). The version number *n* helps you identify the instrumentation version you use in an experiment. The experiment results directory will have a matching version number. The instrument directory is the current working directory; it can be set from the **Admin** menu.

5. Click **OK**.

   This executes the instrumentation process. If there are no problems, the dialog box closes and the message `Instrumentation succeeded` displays in the status area with the version number created.

Making a Test

A *test* defines the program and arguments to be run, the instrumentation criteria, and descriptive information about the test.

Procedure 4-3 Making a Test

1. Select **Make Test** from the **Test** menu.

   This creates a test directory. Figure 4-3 shows the **Make Test** window.

   You specify the name of the test directory in the **Test Name** field, in this case `test0000`. The field displays a default directory `test<`*nnnn*`>`, where *nnnn* is 0000 the first time and incremented for subsequent tests. You can edit this field if necessary.

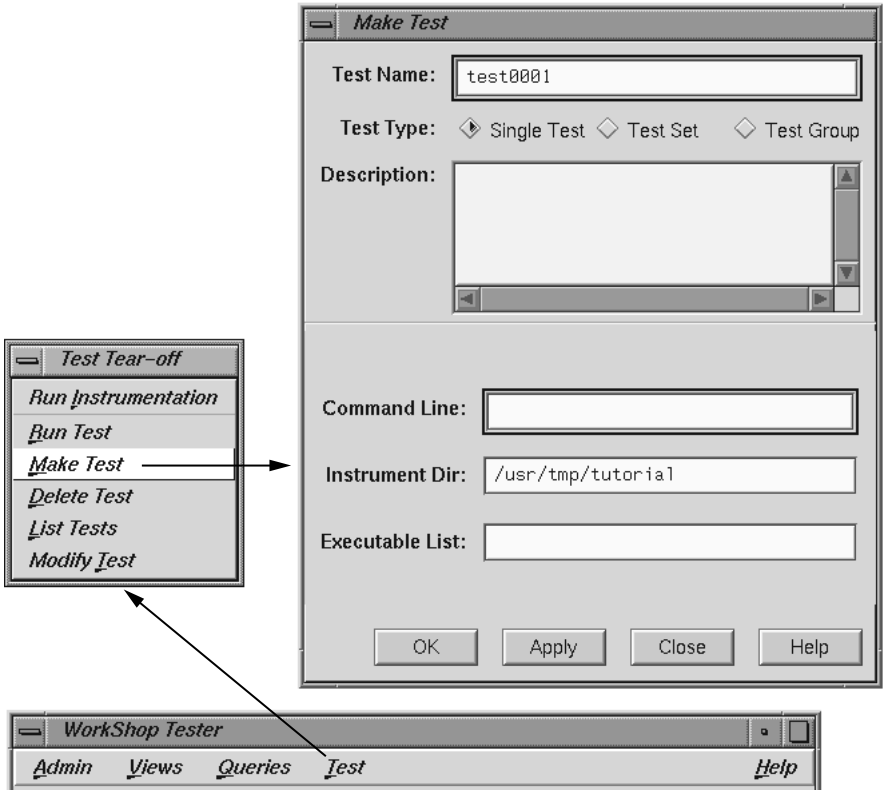

Figure 4-3 Selecting **Make Test**

2. Enter a description of the test in the **Description** field.

   This is optional, but can help you differentiate between tests you have created.

3. Enter the executable to be tested with its arguments in the **Command Line** field, in this example:

   ```
   copyn alphabet targetfile 20
   ```

   This field is mandatory, as indicated by its highlighting.

4. Leave the remaining fields as is.

Tester supplies a default instrumentation directory in the **Instrument Dir** field. The **Executable List** field lets you specify multiple executables when your main program `fork`s, `exec`s, or `sproc`s other processes.

5. Click **OK** to perform the make test operation with your selections.

   The results of the make test operation display in the status area of the main Tester window.

Running a Test

To run a test, we use technology from the WorkShop Performance Analyzer. The instrumented process is set to run, and a monitor process (`cvmon`) captures test coverage data by interacting with the WorkShop process control server (`cvpcs`).

Procedure 4-4 Running a Test

1. Select **Run Test** from the **Test** menu.

   The dialog box shown in Figure 4-4, page 51, is displayed. You enter the test directory in the **Test Name** field. You can also specify a version of the executable in the **Version Number** field if you do not want to use the latest, which is the default.

   The **Force Run** toggle forces the test to be run again even if a test result already exists. The **Keep Performance Data** toggle retains all the performance data collected in the experiment. The **Accumulate Results** toggle sums over the coverage data into the existing experiment results. Both **No Arc Data** and **Remove Subtest Expt** toggles retain less data in the experiments and are designed to save disk space.

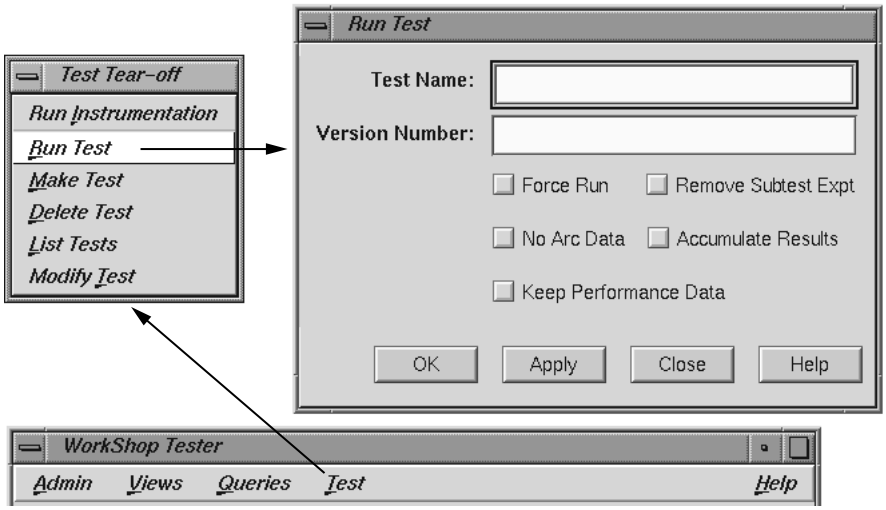

Figure 4-4 Run Test Dialog Box

2. Enter **`test0000`** in the **Test Name** field.
3. Click **OK** to run the test with your selections.

   When the test completes, a status message showing completion displays and you will have data to be analyzed. You can observe the test as it runs in **Execution View**.

Analyzing the Results

You can analyze test coverage data in many ways. In this tutorial, we will illustrate a simple top-down approach. We will start at the top to get a summary of overall coverage, proceed to the function level, and finally go to the actual source lines.

Having collected all the coverage data, now you can analyze it. You do this through the **Queries** menu in the main Tester window.

Procedure 4-5 Analyzing Test Coverage Data

1. Enter **`test0000`** in the **Test Name** field in the main window and select **List Summary** from the **Queries** menu.

This loads the test and changes the main window display as shown in Figure 4-5, page 53. The query type (in this case, **List Summary**) is indicated above the display area. Column headings identify the data, which displays in columns in the coverage display area. The status area is shortened.

The query-specific fields (in this case, coverage weighting factors) that appear below the control buttons and status area are different for each query type. You can change the numbers and click **Apply** to weight the factors differently. The **Executable List** button brings up the **Target List** dialog box. It displays a list of executables used in the experiment and lets you select different executables for analysis. You can select other experiments from the experiment menu (**Expt**).

List Summary shows the coverage data (number of coverage hits, total possible hits, percentage, and weighting factor) for functions, source lines, branches, arcs, and blocks. The last coverage item is the weighted average, obtained by multiplying individual coverage averages by the weighting factors and summing the products.

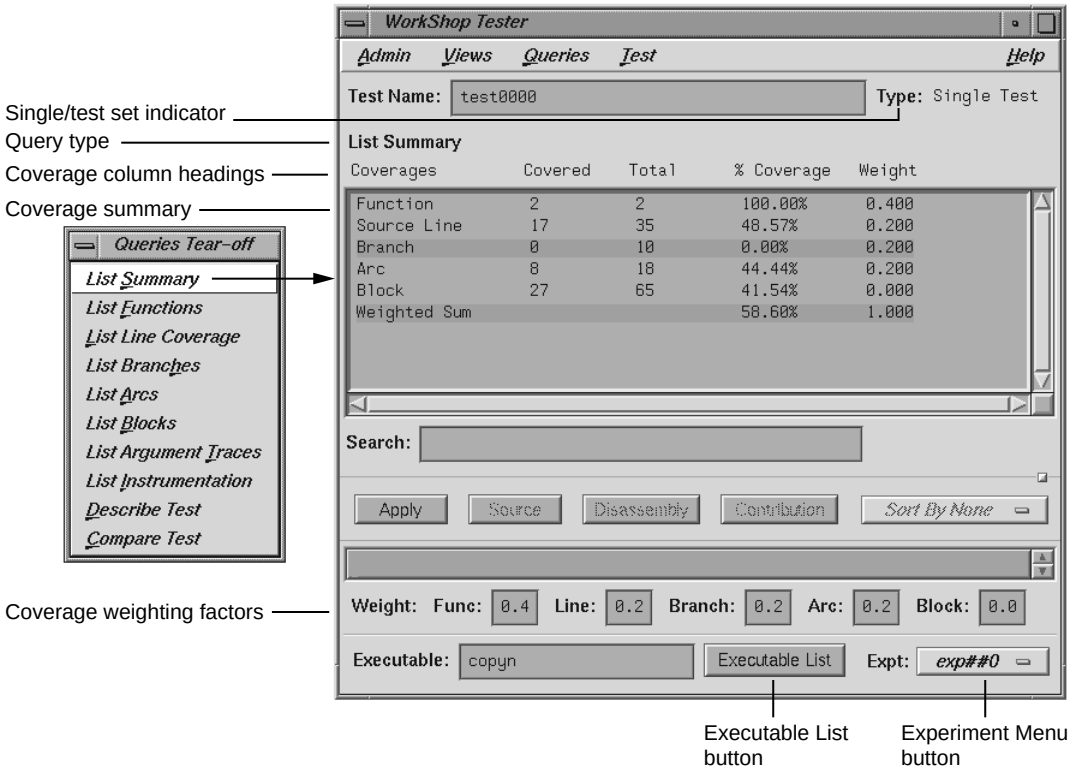

Figure 4-5 List Summary Query Window

2. Select **List Functions** from the **Queries** menu.

   This query lists the coverage data for functions specified for inclusion in this test. The default version is shown in Figure 4-6, page 54, with the available options.

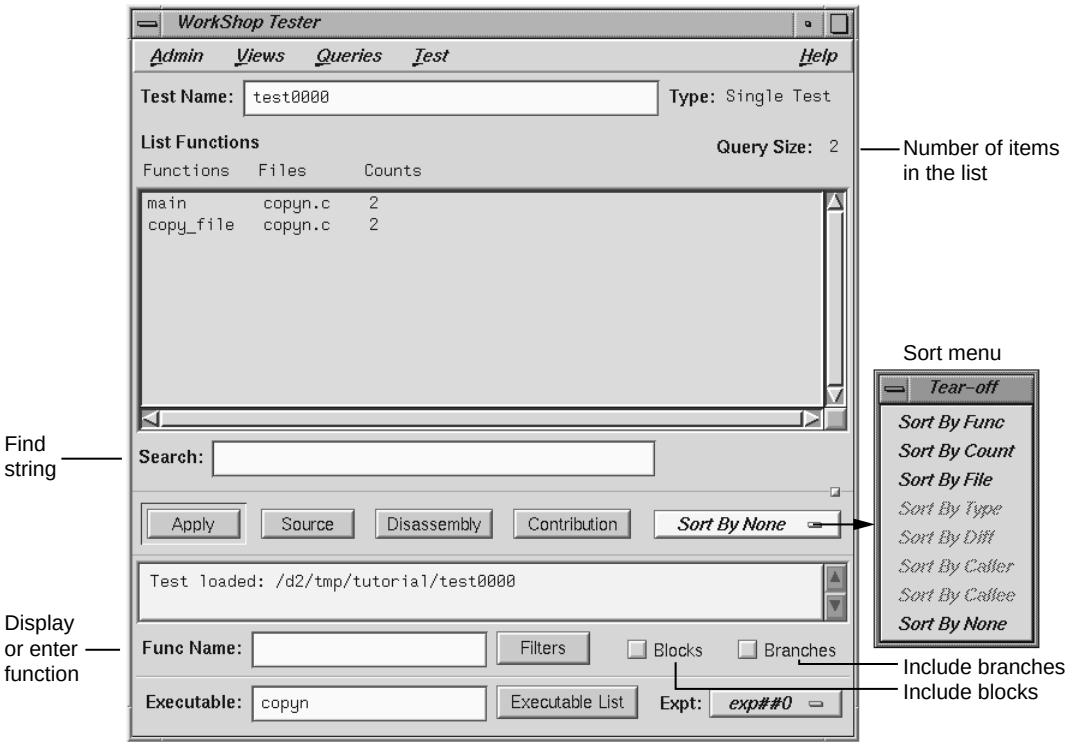

Figure 4-6 List Functions Query with Options

If there are functions with 0 counts, they will be highlighted. The default column headings are **Functions**, **Files**, and **Counts**.

3. Click the **Blocks** and **Branches** toggles.

   The **Blocks** and **Branches** toggle buttons let you display these items in the function list. Figure 4-7, page 55, shows the display area with **Blocks** and **Branches** enabled.

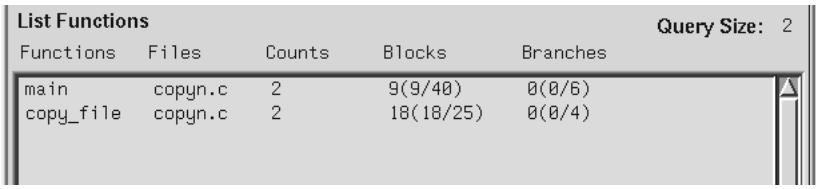

Figure 4-7 List Functions Display Area with **Blocks** and **Branches**

The **Blocks** column shows three values. The number of blocks executed within the function is shown first. The number of blocks covered out of the total possible for that function is shown inside the parentheses. If you divide these numbers, you will arrive at the percentage of coverage.

Similarly, the **Branches** column shows the number of branches covered, followed by the number covered out of the total possible branches. The term *covered* means that the branch has been executed under both true and false conditions.

4. Select the function `main` in the display area and click **Source**.

   The **Source View** window displays with count annotations as shown in Figure 4-8, page 56. Lines with 0 counts are highlighted in the display area and in the vertical scroll bar area. Lines in excluded functions display with no count annotations.

5. Click the **Disassembly** button in the main window.

   The **Disassembly View** window displays with count annotations as shown in Figure 4-9, page 57. Lines with 0 counts are highlighted in the display area and in the vertical scroll bar area.

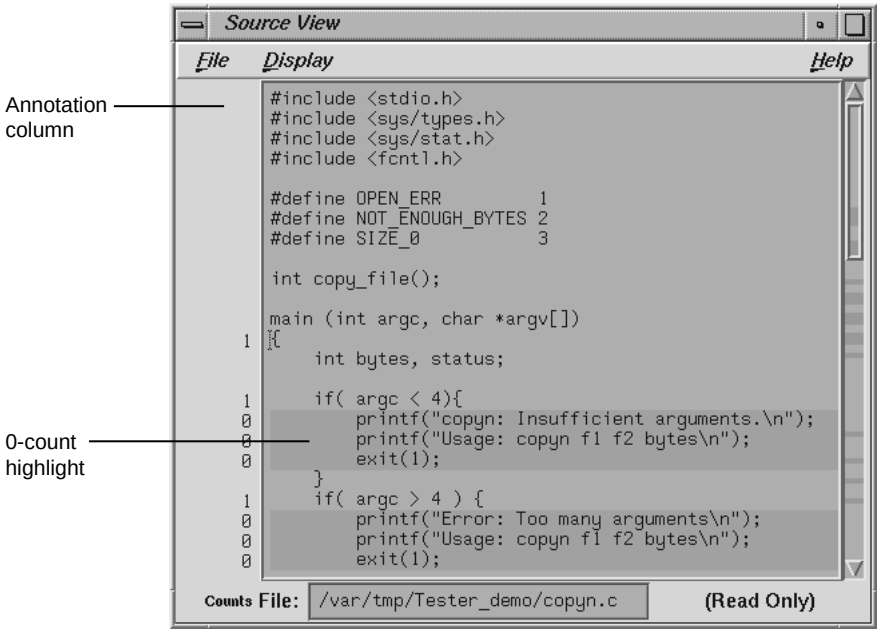

Figure 4-8 Source View with Count Annotations

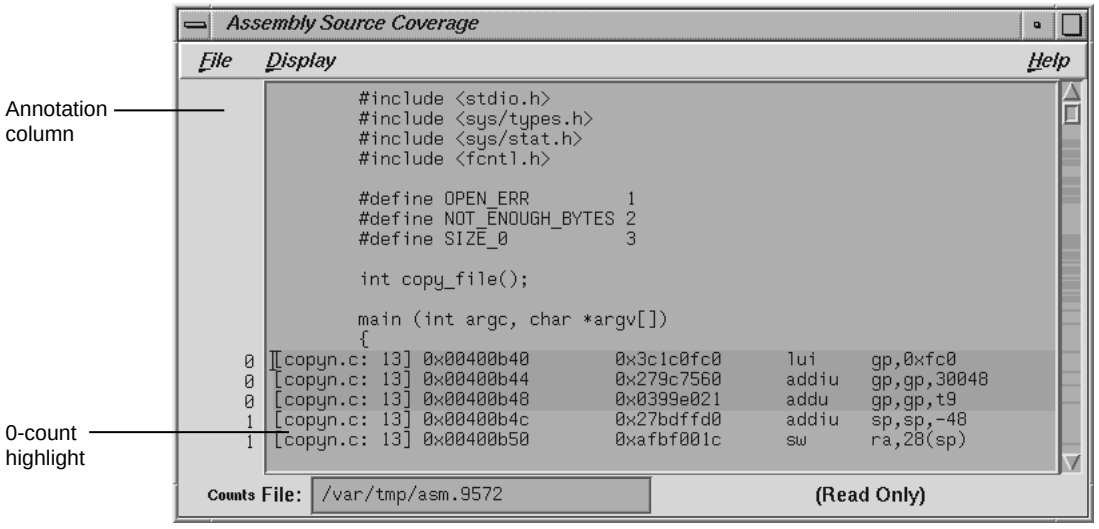

Figure 4-9 Disassembly View with Count Annotations

Tutorial #2: Analyzing a Test Set

In the second tutorial, we are going to create additional tests with the objective of achieving 100% overall coverage. From examining the source code, it seems that the 0-count lines in `main` and `copy_file` are due to error-checking code that is not tested by `test0000`.

Note: This tutorial needs `test0000`, which was created in the previous tutorial.

---

1. Select **Make Test** from the **Test** menu on the Tester main window.

   This displays the **Make Test** dialog box. It is easy to enter a series of tests. Using the **Apply** button in the dialog box instead of the **OK** button completes the task without closing the dialog box. The **Test Name** field supplies an incremented default test name after each test is created.

   We are going to create a test set named `tut_testset` and add to it 8 tests in addition to `test0000` from the previous tutorial. The tests `test0001` and `test0002` pass too few and too many arguments, respectively. `test0003` attempts to copy from a file named `no_file` that does not exist. `test0004`

attempts to pass 0 bytes, which is illegal. `test0005` attempts to copy 20 bytes from a file called `not_enough`, which contains only one byte. In `test0006`, we attempt to write to a directory without proper permission. `test0007` tries to pass too many bytes. In `test0008`, we attempt to copy from a file without read permission.

The following steps show the command line target and arguments and description for the tests in the tutorial. The descriptions are helpful but optional. Figure 4-10 shows the features of the dialog box you will need for creating these tests.

2. Enter **`copyn alphabet target`** in the **Command Line** field, **`not enough arguments`** in the `Description` field, and click **Apply** (or simply press the `Return` key) to make `test0001`.

3. Enter **`copyn alphabet target 20 extra_arg`** in the **Command Line** field, **`too many arguments`** in the `Description` field, and click **Apply** to make `test0002`.

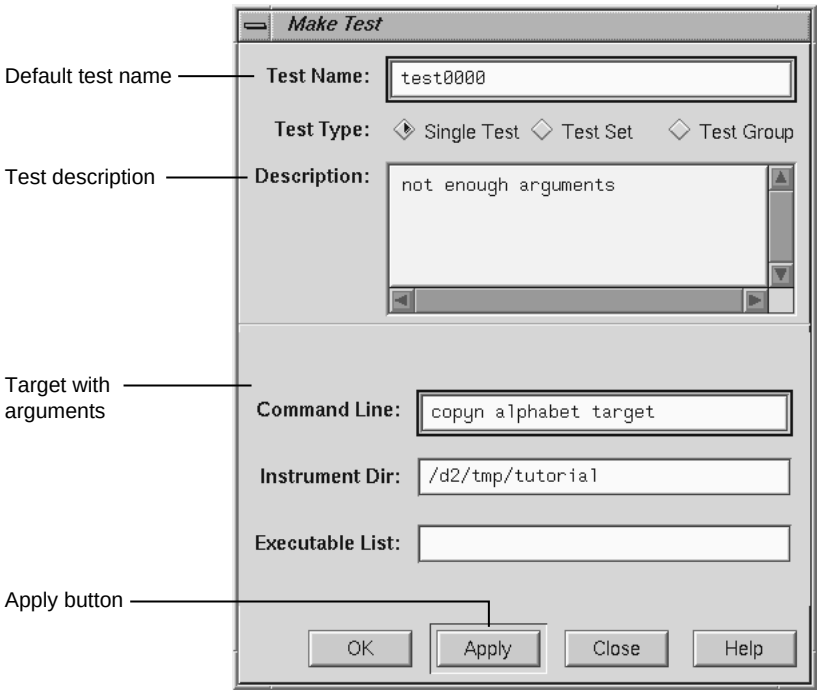

Figure 4-10 Make Test Dialog Box with Features Used in Tutorial

4. Enter **`copyn no_file target 20`** in the **Command Line** field, **`cannot access file`** in the `Description` field, and click **Apply** to make `test0003`.
5. Enter **`copyn alphabet target 0`** in the **Command Line** field, **`pass bad size arg`** in the `Description` field, and click **Apply** to make `test0004`.
6. Enter **`copyn not_enough target 20`** in the **Command Line** field, **`not enough data`** in the `Description` field, and click **Apply** to make `test0005`.
7. Enter **`copyn alphabet /usr/bin/target 20`** in the **Command Line** field, **`cannot create target executable due to permission problems`** in the `Description` field, and click **Apply** to make `test0006`.
8. Enter **`copyn alphabet targetfile 200`** in the **Command Line** field, **`size arg too big`** in the `Description` field, and click **Apply** to make `test0007`.
9. Enter **`copyn /usr/etc/snmpd.auth targetfile 20`** in the **Command Line** field, **`no read permission on source file`** in the `Description` field, and click **Apply** to make `test0008`.

   We now need to create the test set that will contain these tests.

10. Click the **Test Set** toggle in the **Test Type** field.

    This changes the dialog box as shown in Figure 4-11, page 60.

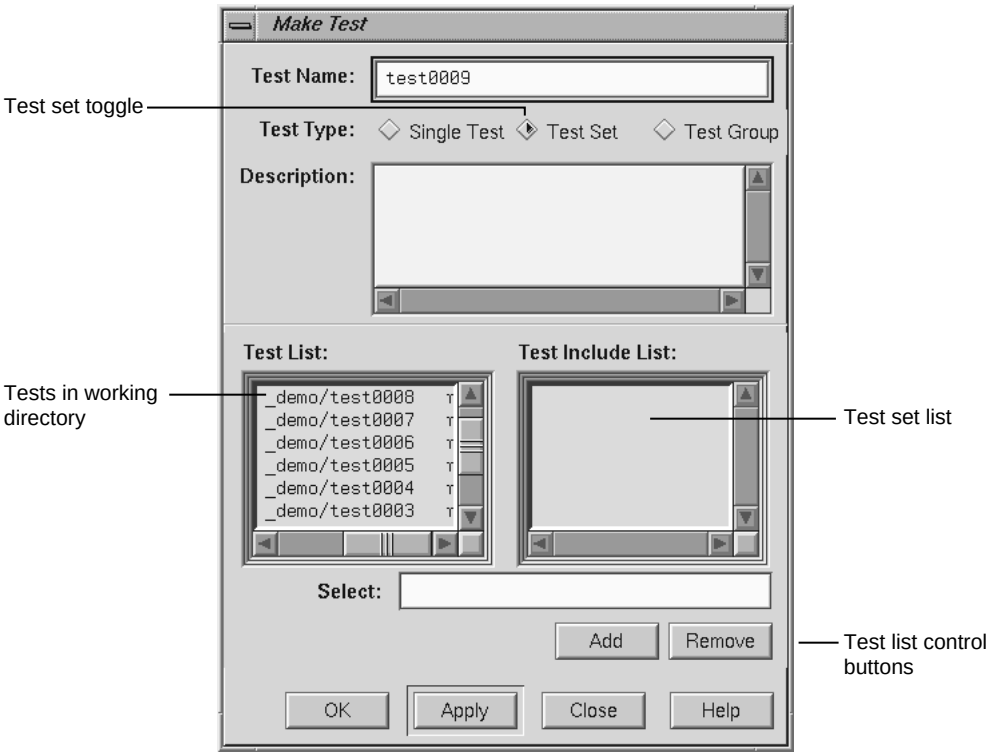

Figure 4-11 Make Test Dialog Box for Test Set Type

11. Change the default in the **Test Name** field to **`tut_testset`**.

    This is the name of the new test set. Now we have to add the tests to the test set.

12. Select the first test in the **Test List** field and click **Add**.

    This displays the selected test in the **Test Include List** field, indicating that it will be part of the test set after you click **OK** (or **Apply** and **Close**).

13. Repeat the process of selecting a test and clicking **Add** for each test in the **Test List** field. When all tests have been added to the test set, click **OK**.

    This saves the test set as specified and closes the **Make Test** dialog box.

14. Enter **`tut_testset`** in the **Test Name** field and select **Describe Test** from the **Queries** menu on the main Tester screen.

This displays the test set information in the display area of the main window.

15. Select **Run Test** from the **Test** menu, enter **`tut_testset`** in the **Test Name** field in the **Run Test** dialog box.

    This runs all the tests in the test set.

16. Make sure **`tut_testset`** is in the **Test Name** field in the main Tester window and select **List Summary** from the **Queries** menu.

    This displays a summary of the results for the entire test set.

17. Select **List Functions** from the **Queries** menu.

    This step serves two purposes. It enables the **Source** button so that we can look at counts by source line. It displays the list of functions included in the test, from which we can select functions to analyze.

18. Click the `main` function, which is displayed in the function list, and click the **Source** button.

    This displays the source code, with the counts for each line shown in the annotations column. Note that the counts are higher now and full coverage has been achieved at the source level (although not necessarily at the assembly level).

Tutorial #3: Exploring the Graphical User Interface

The rest of this chapter shows you how to use the graphical user interface (GUI) to analyze test data. The GUI has all the functionality of the command line interface and in addition shows the function calls, blocks, branches, and arcs graphically.

For a discussion of applying Tester to test set optimization, refer to "Tutorial #3: Optimizing a Test Set", page 27. Although this is written for the command line interface, you can use the graphical interface to follow the tutorial.

1. Enter **`test0000`** in the **Test Name** field of the main window and press the `Enter` key.

   Since `test0000` has incomplete coverage, it is more useful for illustrating how uncovered items appear.

2. Select **List Functions** from the **Queries** menu.

   The list of functions displays in the text view format.

3. Select **Call Tree View** from the **Views** menu.

   The Tester main window changes to call graph format. Figure 4-12, page 62, shows a typical call graph. Initially, the call graph displays the `main` function and its immediate callees.

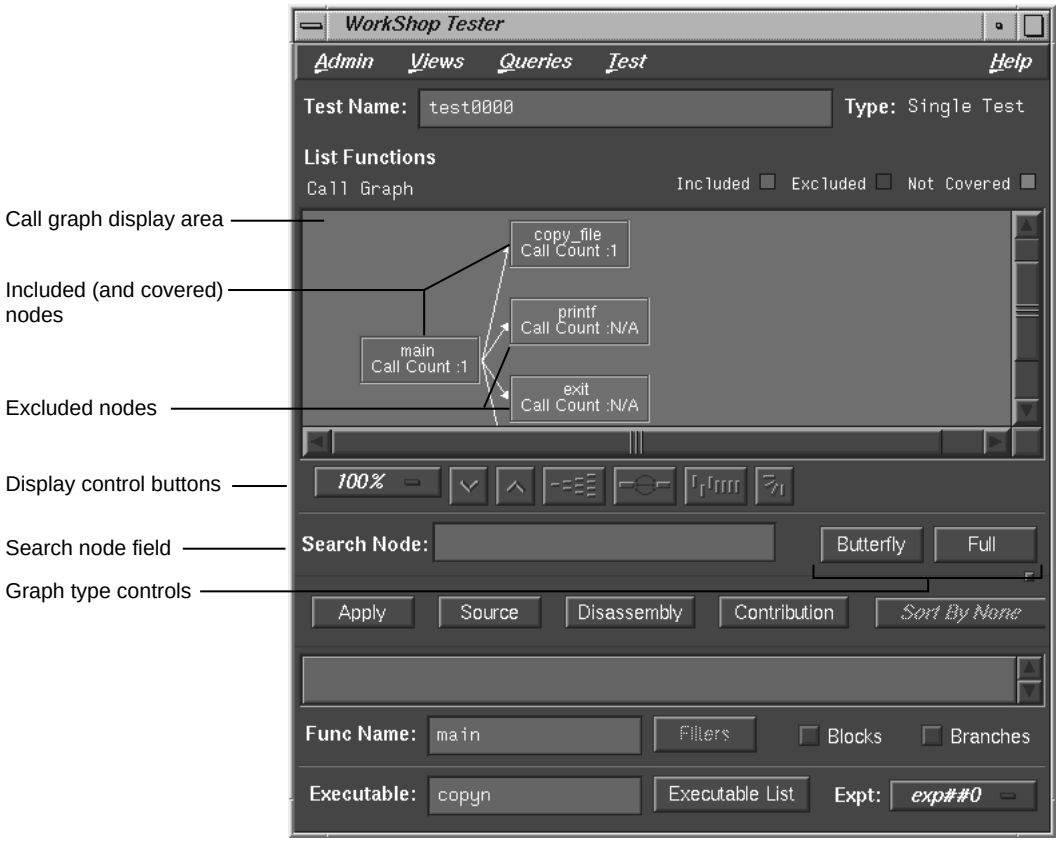

Figure 4-12 Call Graph for **List Functions** Query

The call graph displays functions as nodes and calls as connecting arrows. The nodes are annotated by call count information. Functions with 0 counts are highlighted. Excluded functions when visible appear in the background color.

The controls for changing the display of the call graph are just below the display area (see Figure 4-13, page 63).

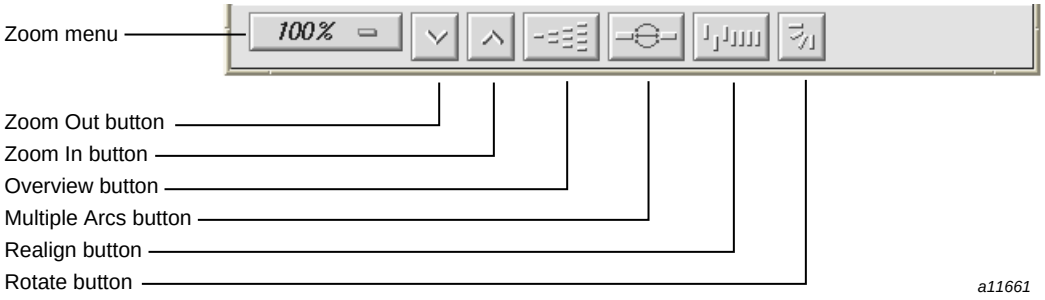

Figure 4-13 Call Graph Display Controls

These facilities are:

- **Zoom menu** icon: shows the current scale of the graph. If clicked on, a popup menu appears displaying other available scales. The scaling range is between 15% and 300% of the nominal (100%) size.
- **Zoom Out** icon: resets the scale of the graph to the next (available) smaller size in the range.
- **Zoom In** icon: resets the scale of the graph to the next (available) larger size in the range.
- **Overview** icon: invokes an overview popup display that shows a scaled-down representation of the graph. The nodes appear in the analogous places on the overview popup, and a white outline may be used to position the main graph relative to the popup. Alternatively, the main graph may be repositioned with its scroll bars.
- **Multiple Arcs** icon: toggles between single and multiple arc mode. Multiple arc mode is extremely useful for the **List Arcs** query, because it indicates graphically how many of the paths between two functions were actually used.
- **Realign** icon: redraws the graph, restoring the positions of any nodes that were repositioned.
- **Rotate** icon: flips the orientation of the graph between horizontal (calling nodes at the left) and vertical (calling nodes at the top).

Entering a function in the **Search Node** field scrolls the display to the portion of the graph in which the function is located.

There are two buttons controlling the type of graph. Entering a node in the **Func Name** field and clicking **Butterfly** displays the calling and called functions for that node only (**Butterfly** mode is the default). Selecting **Full** displays the entire call graph (although not all portions may be visible in the display area).

4. Select **List Arcs** from the **Queries** menu.

   The **List Arcs** query displays coverage data for calls made in the test. Because we were just in call graph mode for the **List Functions** query, **List Arcs** comes up in call graph rather than text mode.

   See Figure 4-14, page 65. To improve legibility, this figure has been scaled up to 150% and the nodes moved by middle-click-dragging the outlines. Arcs with 0 counts are highlighted in color. Notice that in **List Arcs**, the arcs rather than the nodes are annotated.

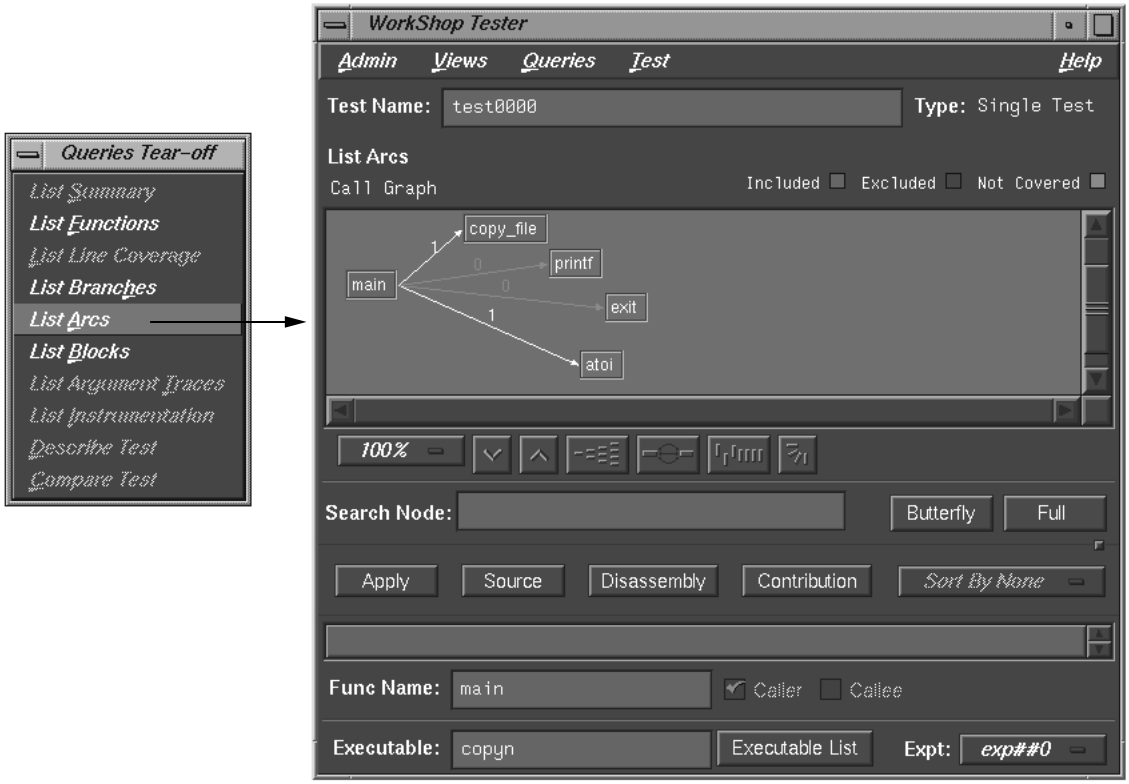

Figure 4-14 Call Graph for **List Arcs** Query

5. Click the **Multiple Arcs** button (the third button from the right in the row of display controls).

   This displays each of the potential arcs between the nodes. See Figure 4-15, page 66. Arcs labeled N/A connect excluded functions and do not have call counts.

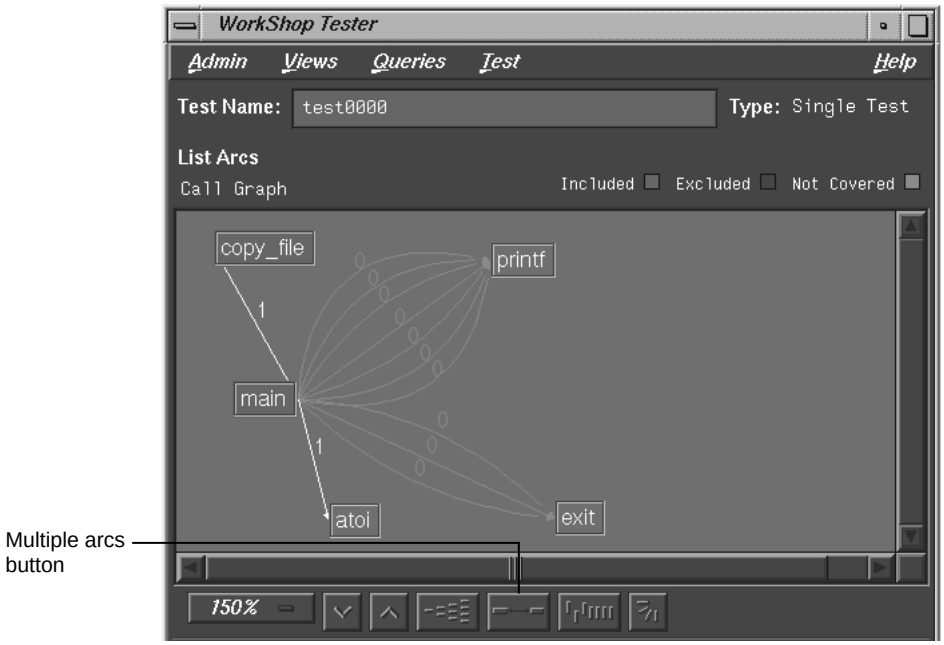

Figure 4-15 Call Graph for **List Arcs** Query — Multiple Arcs

6. Select **Text View** from the **Views** menu.

   This returns the display area to text mode from call graph mode. See Figure 4-16, page 67.

   The **Callers** column lists the calling functions. The **Callees** column lists the functions called. **Line** provides the line number where the call occurred; this is particularly useful if there are multiple arcs between the caller and callee. The **Files** column identifies the source code file. **Counts** shows the number of times the call was made.

   You can sort the data in the **List Arcs** query by count, file, caller, or callee.

Figure 4-16 Test Analyzer Queries: **List Arcs**

7. Select **List Blocks** from the **Queries** menu.

   The window should be similar to Figure 4-17, page 68. The data displays in order of blocks, with the starting and ending line numbers of the block indicated. Blocks that span multiple lines are labeled sequentially in parentheses. The count for each block is shown with 0-count blocks highlighted.

Caution: Listing all blocks in a program may be very slow for large programs. To avoid this problem, limit your **List Blocks** operation to a single function.

Figure 4-17 Test Analyzer Queries: **List Blocks**

You can sort the data for **List Blocks** by count, file, or function.

8. Select **List Branches** from the **Queries** menu.

   The **List Branches** query displays a window similar to Figure 4-18, page 69.

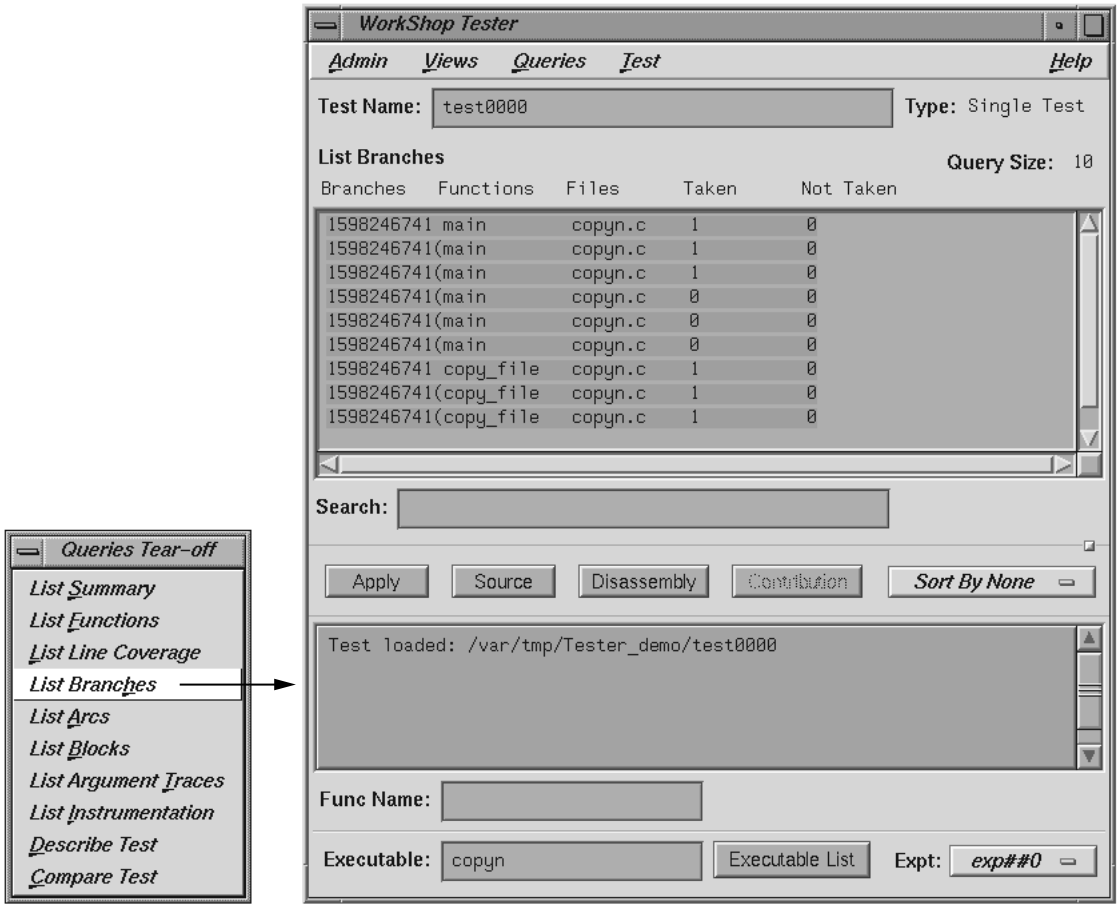

Figure 4-18 Test Analyzer Queries: **List Branches**

The first column shows the line number in which the branch occurs. If there are multiple branches in a line, they are labeled by order of appearance within trailing parentheses. The next two columns indicate the function containing the branch and the file. A branch is considered covered if it has been executed under

both true and false conditions. The **Taken column** indicates the number of branches that were executed only under the true condition. The **Not Taken** column indicates the number of branches that were executed only under the false condition.

The **List Branches** query permits sorting by function or file.

Tester Graphical User Interface Reference

This chapter describes the Tester graphical user interface. It contains these sections:

When you run `cvxcov`, the main Tester window opens and an iconized version of the Execution View appears on your screen. It displays the output and status of a running program and accepts input. To open a closed Execution View, see “Clone Execution View” in "Admin Menu Operations", page 101.

Accessing the Tester Graphical Interface

There are two methods of accessing the Tester graphical user interface:

- Type **`cvxcov`** at the command line with these optional arguments: `-testname` *test* to load the test; `-ver` to show the Tester release version; and `-scheme` *schemename* to set a predefined color scheme.
- Select **Tester** from the **Launch Tool** submenu in a WorkShop **Admin** menu (see Figure 5-1, page 72). The major WorkShop tools, the Debugger, Static Analyzer, and Build Manager provide **Admin** menus from which you can access Tester.

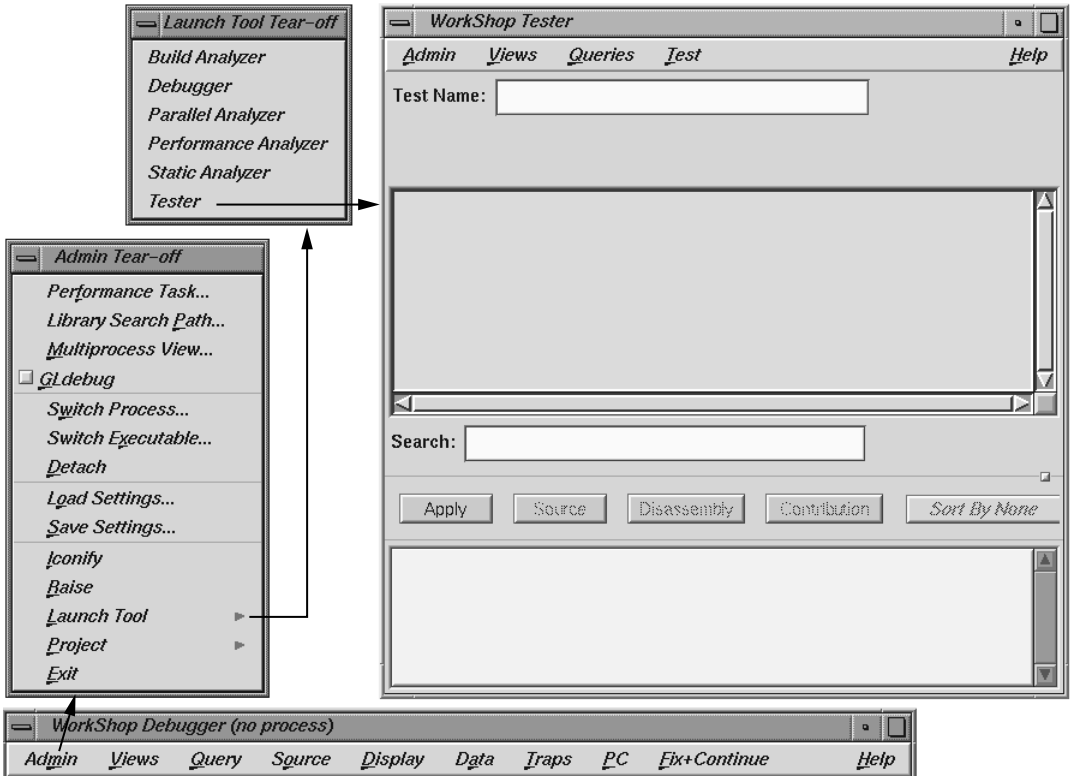

Figure 5-1 Accessing Tester from the WorkShop Debugger

Main Window and Menus

The main window and its menus are shown in Figure 5-2, page 73.

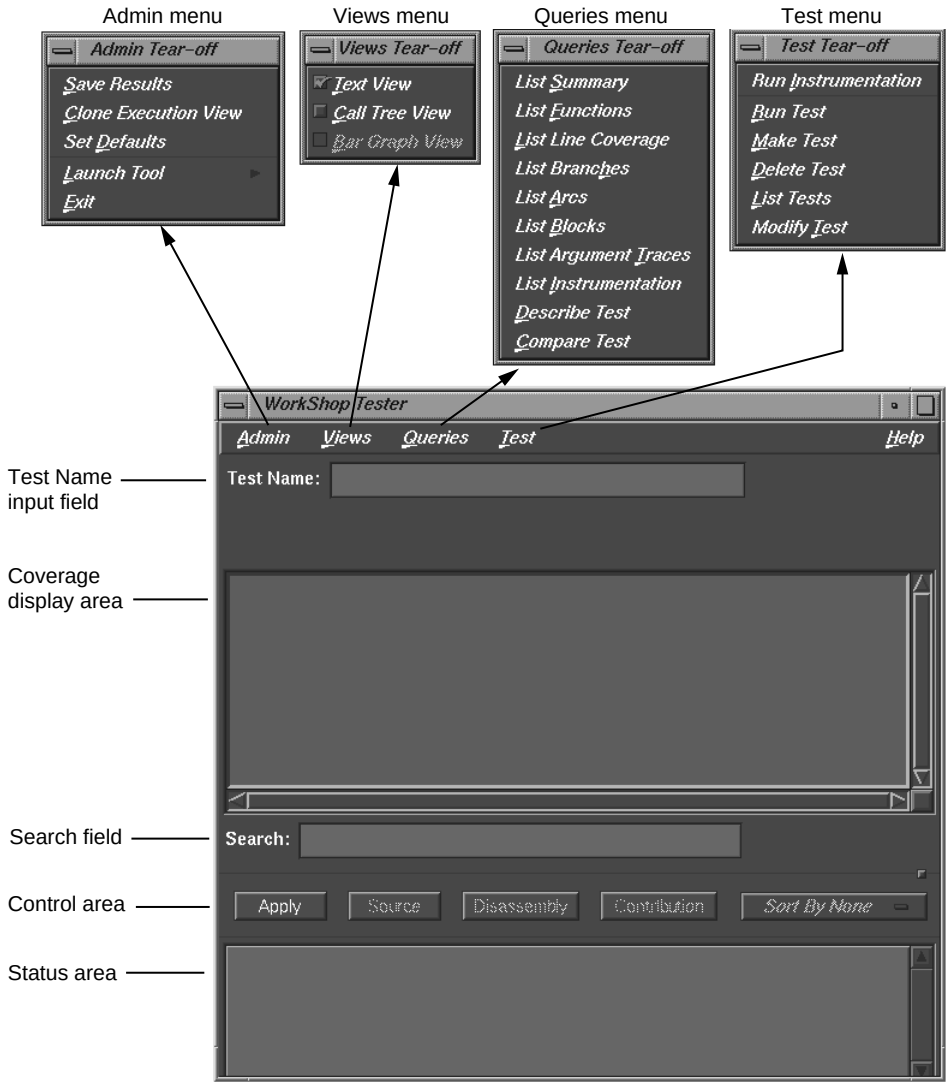

Figure 5-2 Main Test Analyzer Window

Test Name Input Field

The current test is entered (and displayed) in the **Test Name** field. You can switch to a different test, test set, or test group through this field. To the right, the **Type** field indicates whether it is a Single Test, Test Set, or Test Group. You can select a test (test set or test group) from the **List Tests** dialog box under the **Test** menu, to appear in the **Test Name** field in the main window.

Coverage Display Area

Test results display in the coverage display area. You select the results by choosing an item from the **Queries** menu. You can select the format of the data—text, call tree, or bar chart— from the **Views** menu. (Note that the **Text View** format is available for all queries, whereas the other two views are limited.)

The **Query Type** displays under the **Test Name** field, just over the display. It is followed on the far right of the window by the **Query Size** (number of items in the list). Headings above the display are specific to each query.

Search Field

The **Search** field lets you look for strings in the coverage data. It uses an incremental search, that is, as you enter characters, the highlight moves to the first matching target. When you press the `Return` key, the highlight moves to the next occurrence.

Control Area Buttons

The **Apply** button is a general-purpose button for terminating data entry in text fields; you can use the `Return` key equivalently. Both start the query.

The **Source** button lets you bring up the standard **Source View** window with Tester annotations. **Source View** shows the counts for each line and highlights lines with 0 counts. By default, **Source View** is shared with other applications. For example, if `cvstatic` performs a search for function `A`, the results of the query overwrite Tester query results that are in the shared Source View. To stop sharing **Source View** with other applications, set the following resource:

```
cvsourceNoShare:   True
```

The **Disassembly** button brings up the **Disassembly View** window, called **Assembly Source Coverage**, which operates at the machine level in a similar fashion to the **Source View**. This view is not shared with other applications.

Note: If a test has very large counts, there may not be enough space in the **Source View** and **Disassembly View** windows to display them. To make more room, increase the *canvasWidth* resource in the `Cvxcov` app-defaults file, `Cvxcov*test*testdata*canvasWidth`.

The **Contribution** button brings up the **Test Contribution** window with the contributions made by each test so that you can compare the results. It is available for the queries **List Functions**, **List Arcs**, and **List Blocks**. When the tests do not fit on one page, multiple pages are used. Use the **Previous Page** and **Next Page** buttons to display all the tests.

The **Sort** button lets you sort the test results by criteria such as function, count, file, type, difference, caller, or callee. The criteria available depend on the current query.

Status Area and Query-Specific Fields

The status area displays status messages that confirm commands, issue warnings, and indicate error conditions. When you enter a test name in the **Test Name** field, the **Func Name** field appears (along with other items) in the status area for use with queries. Entering a function in this field displays the coverage results limited to that function only.

Additional items display in the area below the status area that change when you select commands from the **Queries** menu. These items are specific to the query selected. Some of these items can be used as defaults (see "Queries Menu Operations", page 87).

Main Window Menus

The **Admin** menu lets you perform general housekeeping concerning saving files, setting defaults, changing directories, launching other WorkShop applications, and exiting.

The **Test** menu lets you create, modify, and run tests, test sets, and test groups.

The **Views** menu lets you choose one of the following modes:

- Text mode, which displays results numerically in columns

- Graphical mode, which displays the following:
  - Functions as nodes (rectangles) annotated by results
  - Calls as arcs (connecting arrows)
- Bar graph mode, which displays the summary of a test as a bar graph.

The **Queries** menu lets you analyze the results of tests. The **Help** menu is standard in all tools.

Test Menu Operations

All operations for running tests are accessed from the **Test** menu in the main Tester window. Figure 5-3, page 77, shows the dialog boxes used to perform test operations.

The **Test** menu provides the following selections:

- **Run Instrumentation**: instruments the target executable. Instrumentation adds code to the executable to collect coverage data. For a more detailed discussion of instrumentation and instrument files, see "Single Test Analysis Process", page 5.

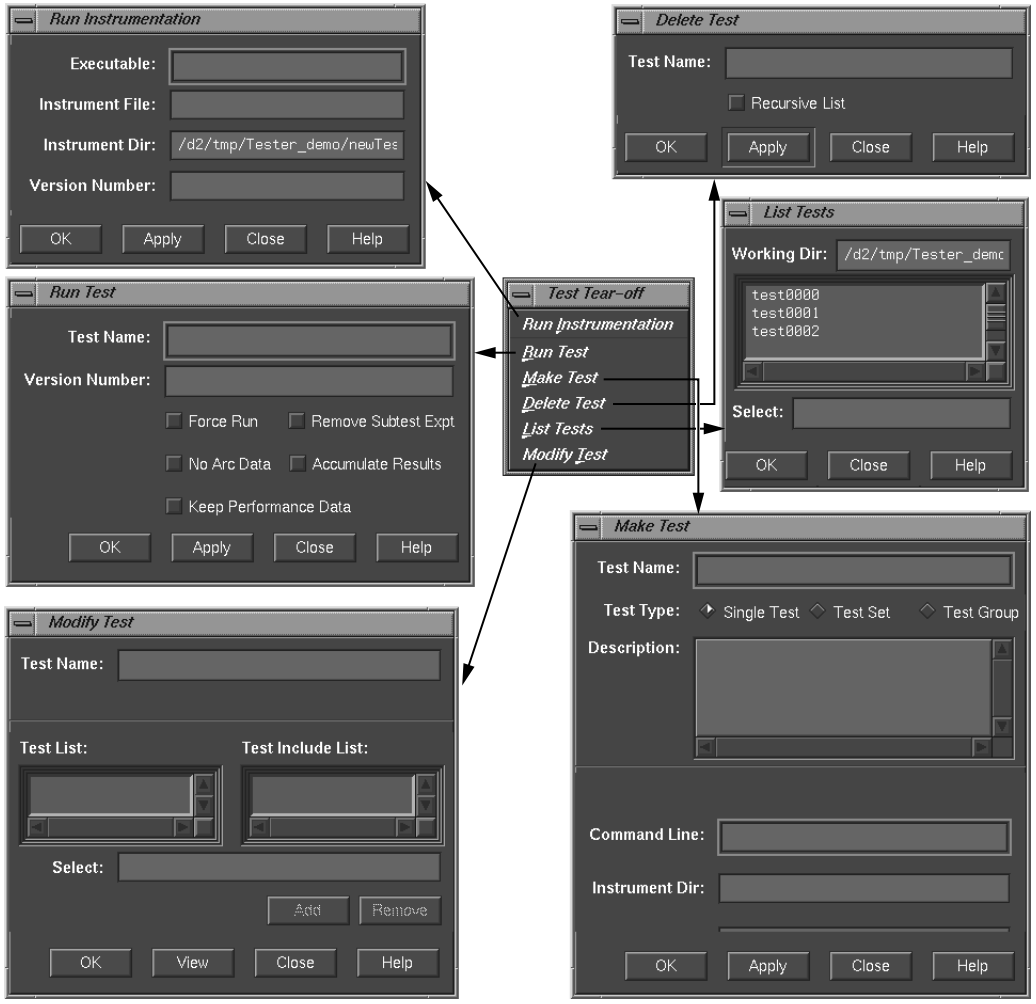

Figure 5-3 Test Menu Commands

The **Run Instrumentation** dialog box (see Figure 5-4, page 78) has these fields:

- **Executable** lets you enter the name of the target.
- **Instrumentation File** is for entering the instrumentation file, which is an ASCII description of the instrumentation criteria for the experiment.

– **Instrumentation Dir** lets you enter the directory in which the instrumentation file is stored (not necessary if you are using the current working directory).

– **Version Number** lets you specify the version number of the instrumentation directory (**ver##<versionnumber>**). If this field is left blank, the version number increments automatically.

If you are testing multiple executables (that is, testing coverage of an executable that `fork`s, `exec`s, or `sproc`s other processes), then you need to store these in the same instrumentation directory. You do this by entering the same number in the **Version Number** field.

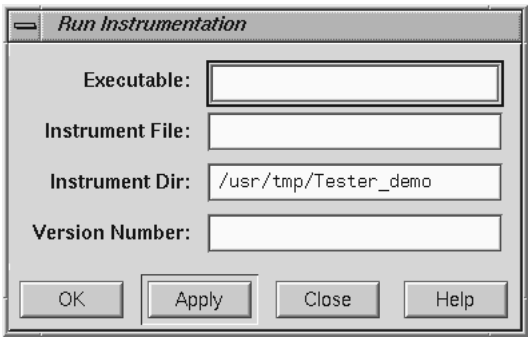

Figure 5-4 Run Instrumentation Dialog Box

- **Run Test**: invokes the executable with selected arguments and collects the coverage data. The **Run Test** dialog box (see Figure 5-5, page 79) provides these fields and buttons:

  – **Test Name** is for entering the test name.

  – **Version Number** is for entering the version number of the directory (`ver## <number>`) containing the instrumented executable. If you are using the most current (highest) version number, then you can leave the field blank; otherwise, you need to enter the desired number.

  – **Force Run** is a toggle that when turned on causes the test to be run even if results already exist.

  – **Keep Performance Data** is a toggle that when turned on retains all the performance data collected in the experiment.

- **Accumulate Results** is a toggle that when turned on accumulates (sums over) the coverage data into the existing experiment results.
- **No Arc Data** prevents arc information from being collected in the experiment. It cannot be used with **List Arcs** or a **Call Tree View**. **List Summary** and **Compare Test** will have 0% coverage on arc items. Use it to save space if you do not need arc data.
- **Remove Subtest Expt** removes results for individual subtests for test sets or test groups, letting you see the top level and taking less space. There will be no data to query if you are querying a subtest.

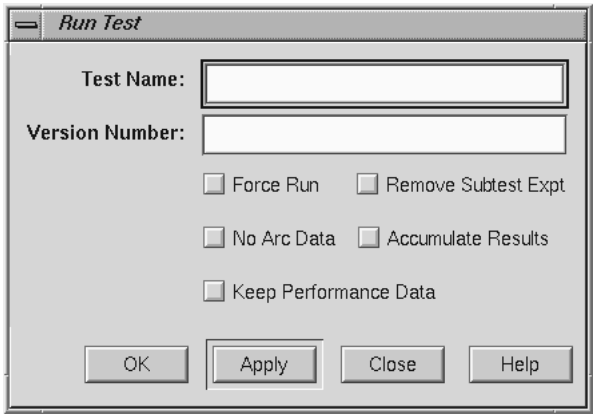

Figure 5-5 Run Test Dialog Box

- **Make Test**: creates a test directory where the coverage data is to be stored and stores a TDF (test description file).

  The **Make Test** dialog box (see Figure 5-6, page 80) provides these fields for tests, test sets, and test groups:

  - **Test Name** is for entering the test name.
  - **Test Type** is a toggle for indicating the type of test: single, test set, or test group (for dynamically shared objects).
  - **Description** lets you enter a description to document the test.

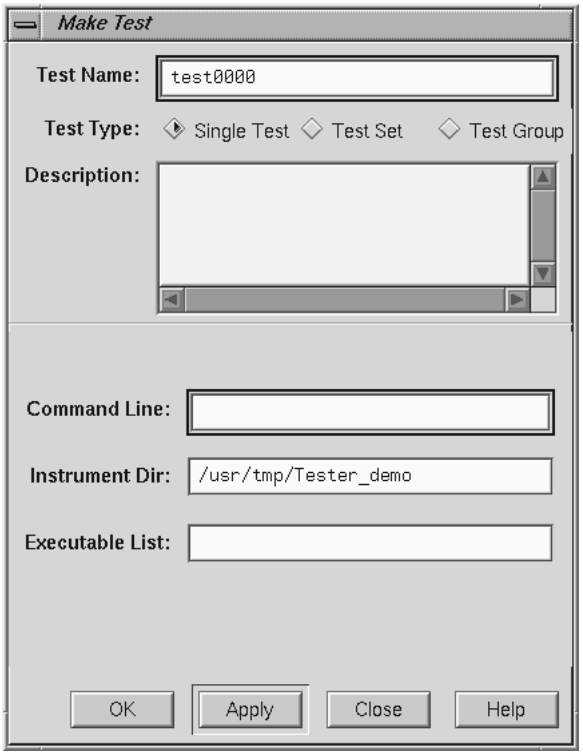

Figure 5-6 Make Test Dialog Box

If you select **Single Test**, the following fields are provided:

- **Command Line** lets you enter the target and any arguments to be used in the test.
- **Instrument Dir** is the directory in which the instrumentation file and related data are stored (not necessary if current working directory).
- **Executable List** is used if you are testing coverage of an executable that `fork`s, `exec`s, or `sproc`s other processes and want to include those processes. You must specify these executables in the **Executable List** field.

If you select **Test Set**, the following fields and buttons are provided:

- **Test List** contains all the tests in the working directory.

– **Test Include List** (to the right) displays tests included in the test set or test group.

– **Add** looks at the selected item in the **Test List** or **Select field** and adds it to the **Test Include List**.

– **Remove** looks at the selected item in the **Test Include List** and removes it.

– **Select** displays the currently selected test.

For a test group (see Figure 5-7, page 81), the following field is added to the same fields and buttons used for a test set:

– **Targets** lets you enter a list of target DSOs or shared libraries, separated by spaces.

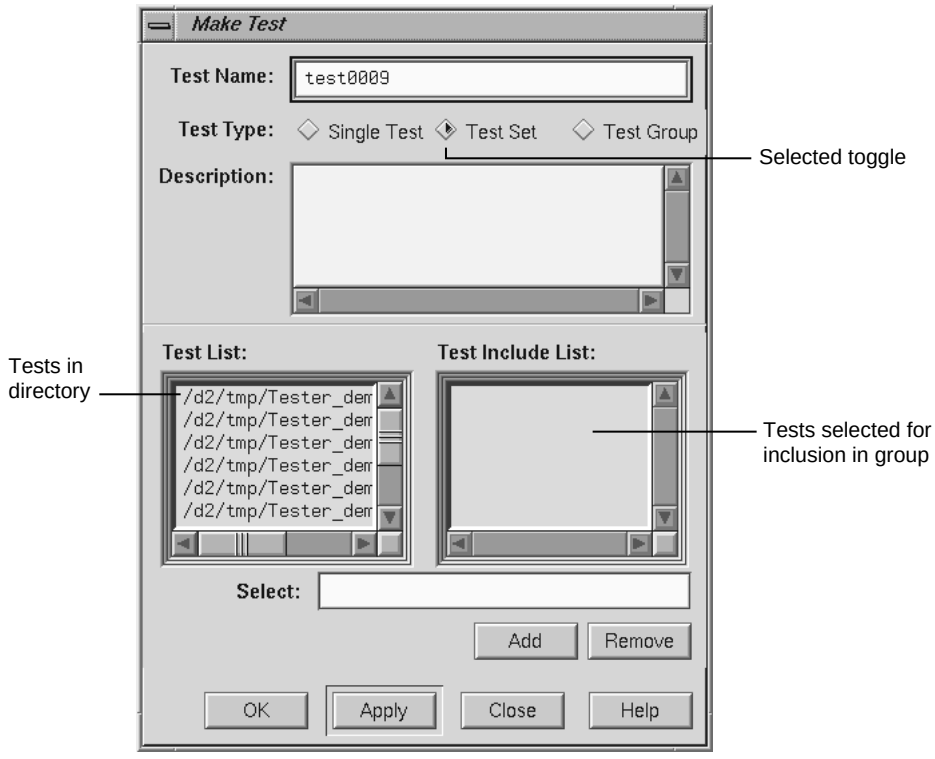

Figure 5-7 Make Test Dialog Box with Test Group Selected

- **Delete Test**: removes the specified test directory and its contents. The **Delete Test** dialog box (see Figure 5-8, page 82) provides these fields:
  - **Test Name** is for entering the test name.
  - **Recursive List** is a toggle that when turned on includes all subtests in the removal of test sets and test groups.

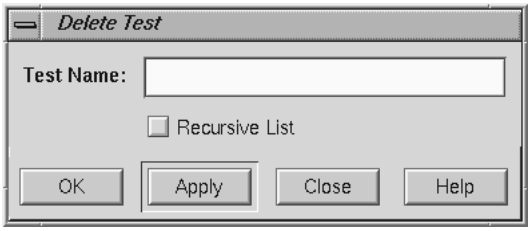

Figure 5-8 Delete Test Dialog Box

- **List Tests**: shows you the tests in the current working directory. The **List Tests** dialog box (see Figure 5-9, page 83) provides these fields:
  - **Working Dir** shows the directory containing the tests.
  - A scrollable list field displays the tests present in the specified directory. The scroll bars let you navigate through the tests if they do not fit completely in the field. Clicking an item places it in the **Select** field. Double-clicking on a test selects and loads it.
  - **Select** displays the test name you type in or that you clicked in the list. Click **OK** to load your selection into the **Test Name** field of the main Tester window.
  - **Close** lets you exit without loading a selection.

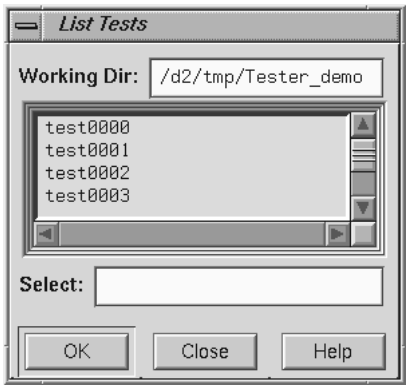

Figure 5-9 List Tests Dialog Box

- **Modify Test**: lets you modify a test set or test group. You enter the test name in the **Test Name** field and press the `Return` key or click the **View** button to load it.

  The **View** button changes to **Apply**, the **Test List** field displays tests in the current working directory, and the **Test Include List** field displays the contents of the test set or test group. You can then add or delete tests, test sets, or test groups in the current test set or test group, respectively. The **Modify Test** dialog box (see Figure 5-10, page 84) has these fields:

  – **Test Name** is for entering the test name.

  – **Test List** displays the tests in the current directory.

  – **Test Include List** displays the subtests for the test specified in the **Test Name** field.

  – **Select** displays the test currently selected for adding or removing. You can enter the test directly in this field instead of selecting it from the **Test List** or **Test Include List**.

  – The **Add** button lets you add the selected test to the **Test Include List**.

  – The **Remove** button lets you delete the selected test from the **Test Include List**.

  – The **Apply** button applies the changes you have selected. (The button name is **View** until you load something.)

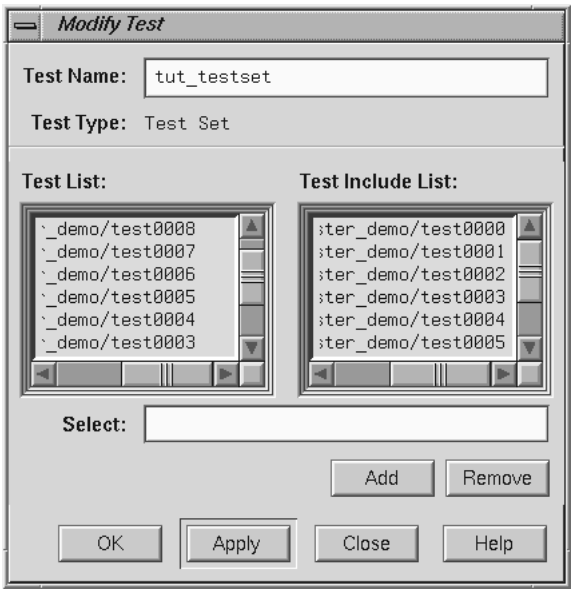

Figure 5-10 Modify Test Dialog Box after Loading Tests

Views Menu Operations

The **Views** menu has three selections that let you view coverage data in different forms. The selections are:

- **Text View**: displays the coverage data in text form. The information displayed depends on which query you have selected. See Figure 5-11, page 85.

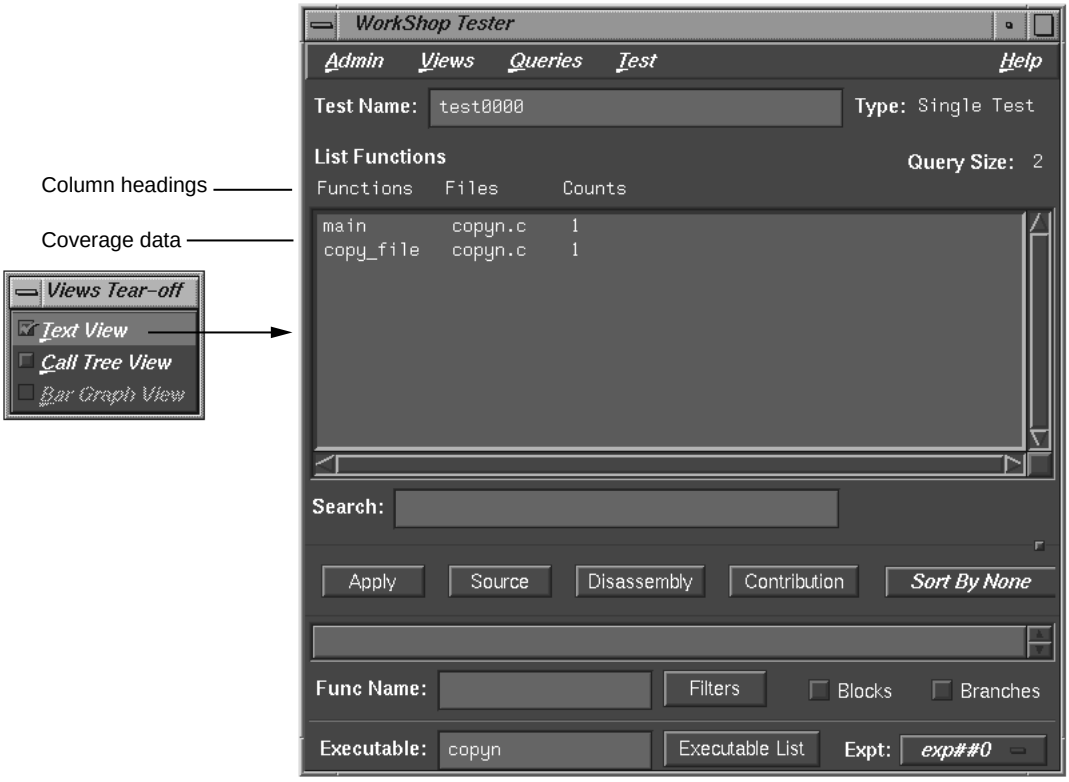

Figure 5-11 List Functions Query in **Text View** Format

- **Call Tree View**: displays coverage data graphically, with functions as nodes (rectangles) and calls as arcs (connecting arrows). This view is only valid for **List Functions**, **List Blocks**, **List Branches**, and **List Arcs**. See Figure 5-12, page 86. It is not available if you run a test with `No Arc Data` on.

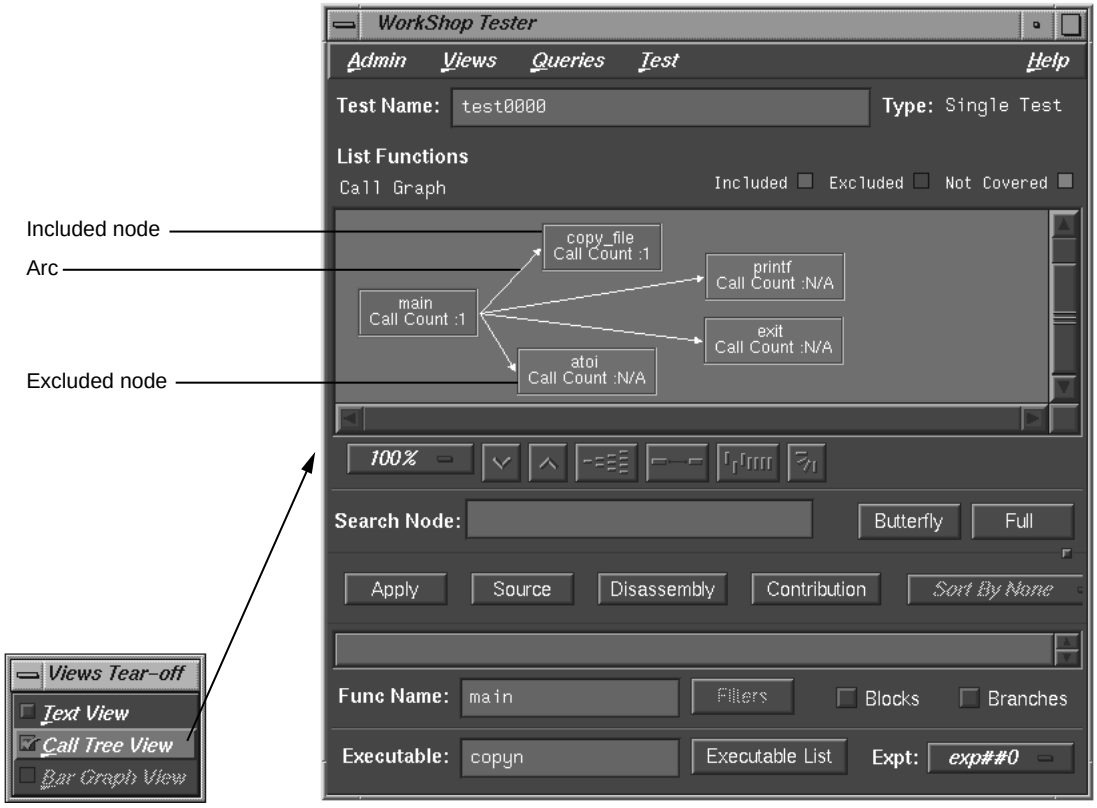

Figure 5-12 List Functions Query in **Call Tree View** Format

- **Bar Graph View**: displays a bar chart showing the percentage covered for functions, lines, blocks, branches, and arcs. See Figure 5-13, page 87. This view is only valid for **List Summary**, which is described in detail in "Queries Menu Operations", page 87.

Figure 5-13 List Summary Query in **Bar Graph View** Format

Queries Menu Operations

The **Queries** menu provides different methods for analyzing the results of coverage tests. Each type of query displays the coverage data in the coverage display area in the main Tester window and displays items that are specific to the query in the area below the status area. When you set these items for a query, the same values are used by default for subsequent queries until you change them. You can set these defaults before the first query or as part of any query. For a single test or test set, all queries except **Describe Test** have the fields shown in Figure 5-15, page 88.

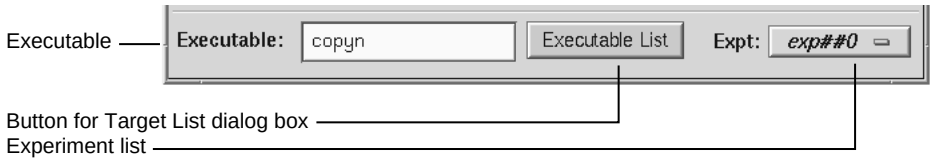

Figure 5-14 Query-Specific Default Fields for a Test or Test Set

The **Executable** field displays the executable associated with the current coverage data. You can switch to a different executable by entering it directly in this field. You can also switch executables by clicking the **Executable List** button, selecting from the list in the **Target List** dialog box and clicking **Apply** in the dialog box.

The experiment menu (**Expt**) lets you see the results for a different experiment that uses the same test criteria.

Note: When you are performing queries on a test group, the **Executable** field changes to **Object** field and the **Executable List** button changes to **Object List** as shown in Figure 5-15, page 88. These items act analogously except that they operate on dynamically shared objects (DSOs).

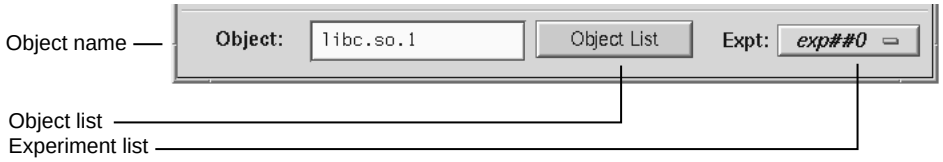

Figure 5-15 Query-Specific Default Fields for a DSO Test Group

The **Queries** menu (see Figure 5-16, page 89) has these selections:

Figure 5-16 Queries Menu

- **List Summary**: shows the overall coverage based on the user-defined weighted average over function, source line, branch, arc, and block coverage. The coverage data appears in the coverage display area. A typical summary appears in Figure 5-17, page 90.

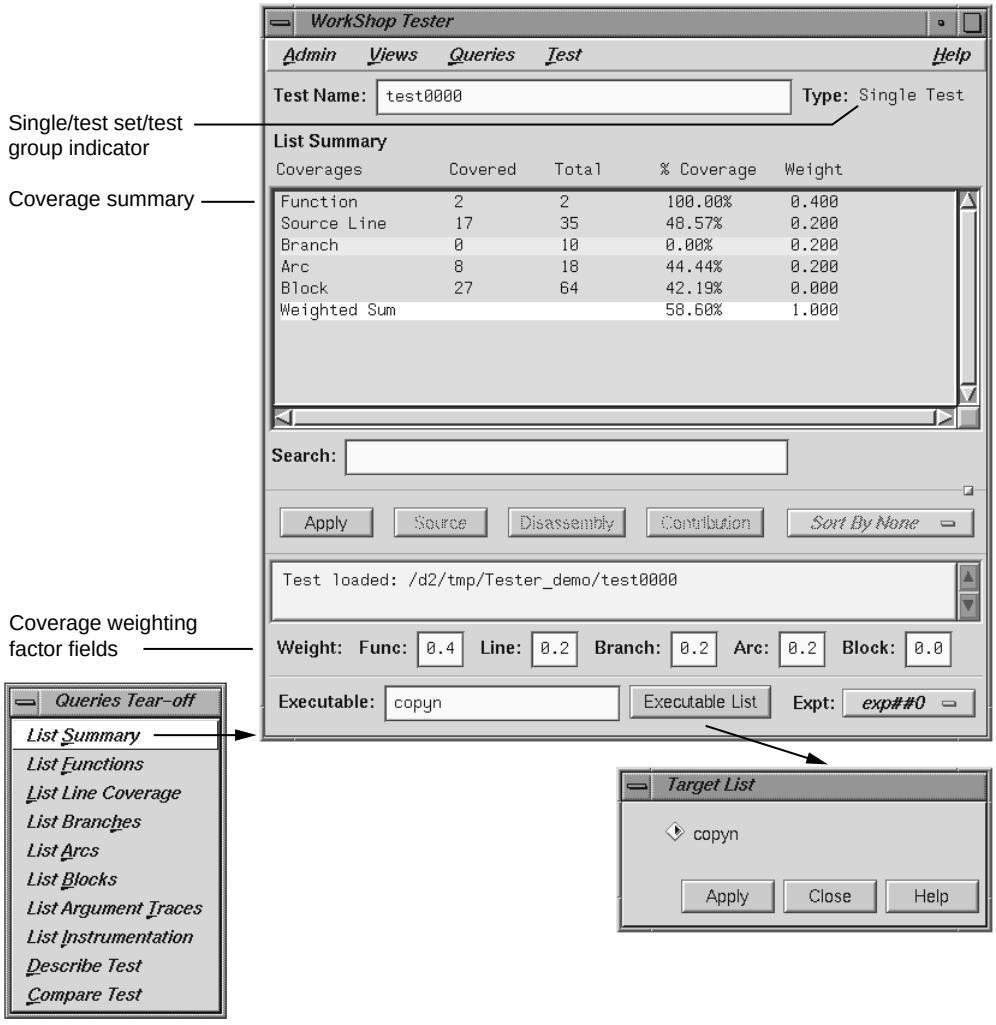

Figure 5-17 List Summary Query

The **Coverages** column indicates the type of coverage. The **Covered** column shows the number of functions, source lines, branches, arcs, and blocks that were executed in this test (or test set or test group). The **Total** column indicates the total number of items that could be executed for each type of coverage. The **% Coverage** column is simply the **Covered** value divided by the **Total** value in each

category. The **Weight** column indicates the weighting assigned to each type of coverage. It is used to compute the **Weighted Sum**, a user-defined factor that can be used to judge the effectiveness of the test. The **Weighted Sum** is obtained by first multiplying the individual coverage percentages by the weighting factors and then summing the products.

The **List Summary** command causes the coverage weighting factor fields to display below the status area. Use these to adjust the factor values as desired. They should add up to 1.0.

If you select **Bar Graph View** from the **Views** menu, the summary will be shown in bar graph format as shown in Figure 5-13, page 87. The percentage covered is shown along the vertical axis; the types of coverage are indicated along the horizontal axis.

- **List Functions**: displays the coverage data for functions in the specified test. The **Functions** column heading identifies the function, **Files** shows the source file containing the function, and **Counts** displays the number of times the function was executed in the test.

  List Functions enables the sort menu that lets you determine the order in which the functions display. Only the sort criteria appropriate for the current query are enabled, in this case, **Sort By Func**, **Sort By Count**, and **Sort By File** as shown in Figure 5-18, page 92.

  The **Search** field scrolls the list to the string entered. The string may occur in any of the columns. This is an incremental search and is activated as you enter characters, scrolling to the first matching occurrence.

  Entering a function in the **Func Name** field displays the coverage results limited to that function only in the display area.

  The **Filters** button displays the **Filters** dialog box, which lets you enter filter criteria to display a subset of the coverage results. There are three types of filters: **Function Count**, **Block Count (%)**, and **Branch Count (%)**.

  For blocks or *branch coverage*, use the toggles described below. Following each label is an operator menu to define the relationship to the limit quantity entered. Each filter type has a text field for entering the desired limit. The limits for **Block Count** and **Branch Count** are percentages (of coverage) and can also be entered using sliders.

Two toggles are available for including branch and block counts. Both appear as actual counts followed by parentheses containing the ratio of counts to total possible.

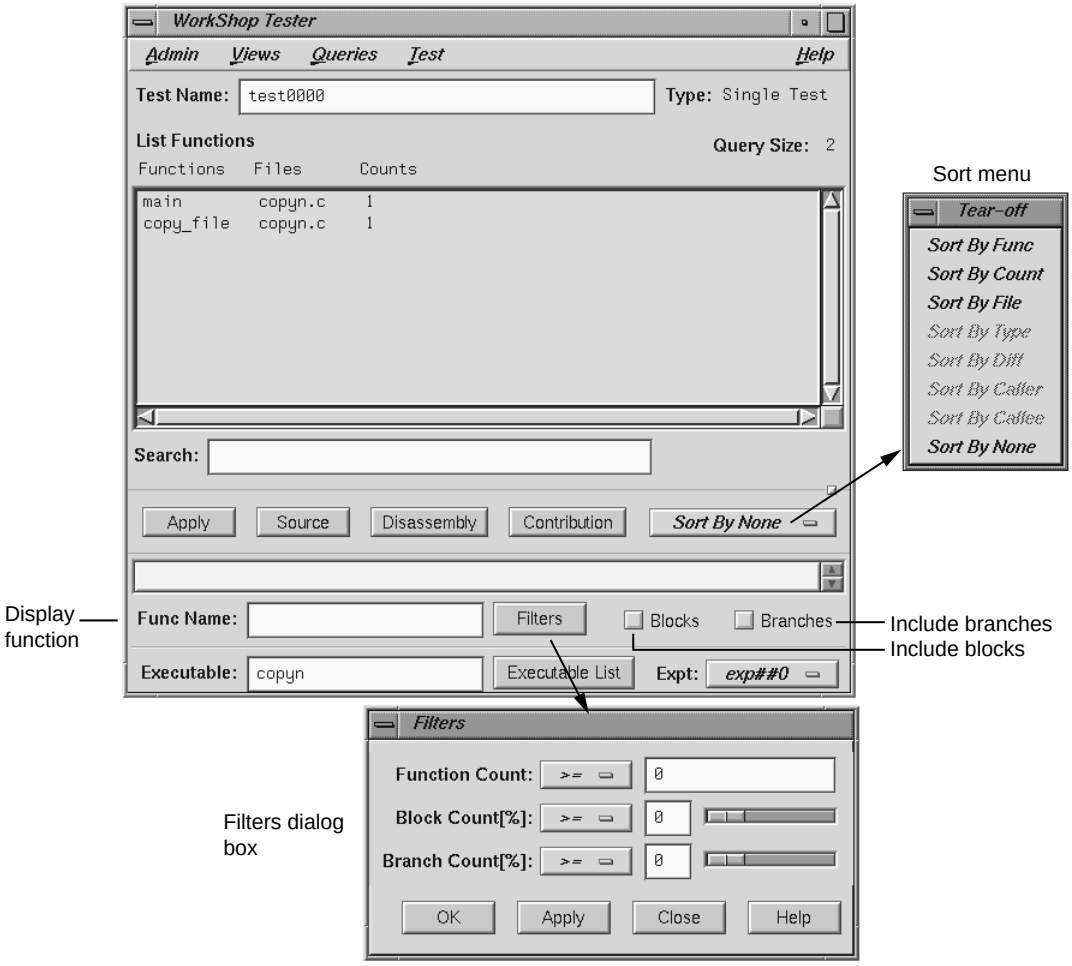

Figure 5-18 List Functions Query with Options

If you select **Call Tree View** from the **Views** menu with a **List Functions** query, a call graph displays (see Figure 5-19, page 93). The call graph displays coverage

data graphically, with functions as nodes (rectangles) and calls as arcs (connecting arrows). The nodes are color-coded according to whether the function was included and covered in the test, included and not covered, or excluded from the test. Arcs labeled N/A connect excluded functions and do not have call counts.

If you hold down the right mouse button over a node, the node menu displays, including the function name, coverage statistics, and standard node manipulation commands. If you have a particularly large graph, you may find it useful to zoom to 15% or 40% and look at the coverage statistics through the node menu.

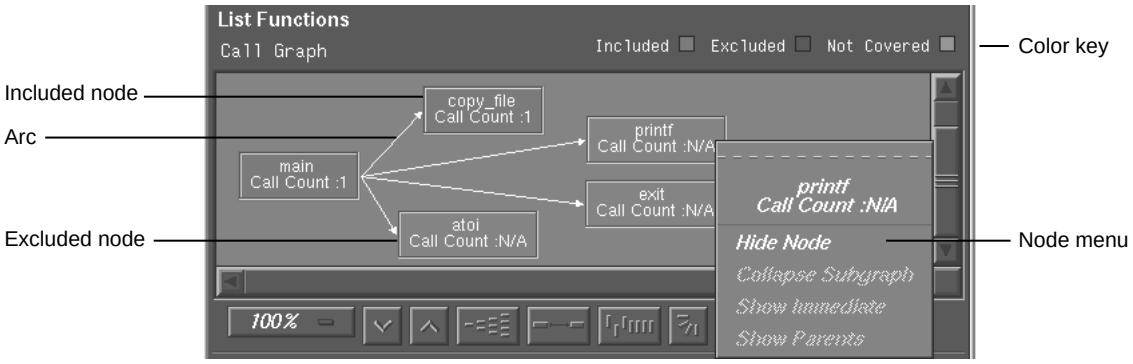

Figure 5-19 List Functions Example in **Call Tree View** Format

- **List Blocks**: displays a list of blocks for one or more functions and the count information associated with each block (see Figure 5-20, page 94). The **Block**s column displays the line number in which the block occurs.

  If there are multiple blocks in a line, blocks subsequent to the first are shown in order with an index number in parentheses. The other three columns show the function and file containing the block and the count, that is, the number of times the block was executed in the test. Uncovered blocks (those containing 0 counts) are highlighted. Block data can be sorted by function, file, or count.

  Be careful before listing all blocks in the program, since this can produce a lot of data. Entering a function in the **Func Name** field displays the coverage results limited to that function only in the display area.

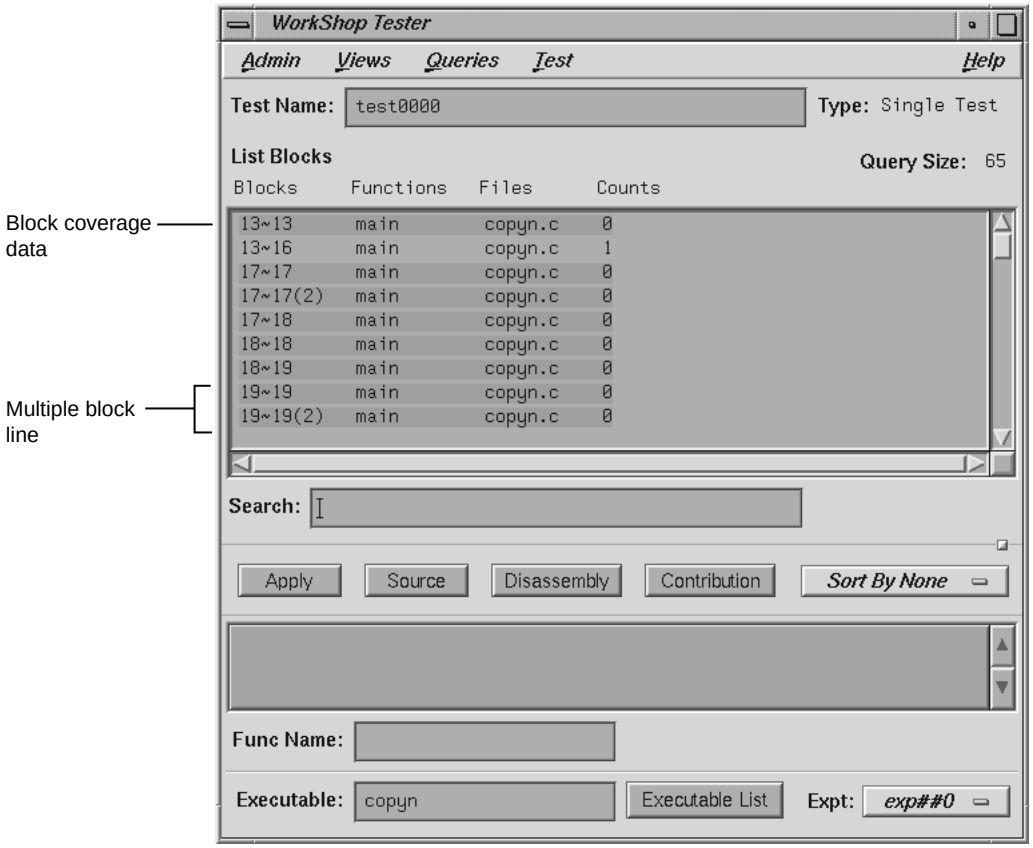

Figure 5-20 List Blocks Example

- **List Branches**: lists coverage information for branches in the program. Branch coverage counts assembly language branch instructions that are taken and not taken. See Figure 5-21, page 95.

  The first column shows the line number in which the branch occurs. If there are multiple branches in a line, they are labeled by order of appearance within trailing parentheses.

  The next two columns indicate the function containing the branch and the file. A branch is considered covered if it has been executed under both true and false conditions. The **Taken** column indicates the number of branches that were

executed only under the true condition. The **Not Taken** column indicates the number of branches that were executed only under the false condition. Branch coverage can be sorted only by function and file.

Entering a function in the **Func Name** field displays the coverage results limited to that function only in the display area.

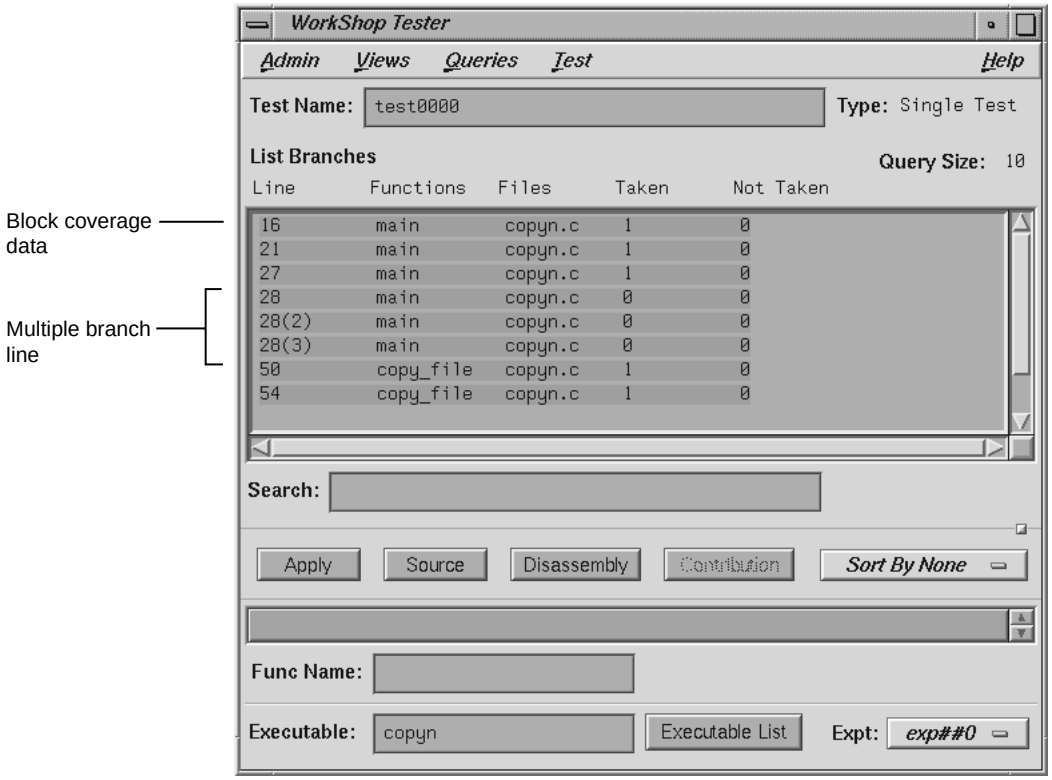

Figure 5-21 List Branches Example

- **List Arcs**: shows arc coverage (that is, the number of arcs taken out of the total possible arcs). An arc is a call from one function (caller) to another (callee). See Figure 5-22, page 96. The caller and callee functions are identified in the first two columns. The **Line** column identifies the line in the caller function where the call occurs. The file and arc execution count display in the last two columns.

Figure 5-22 List Arcs Example

Entering a function in the **Func Name** field displays the coverage results limited to that function only.

The **Caller** and **CalleeFunc Name** toggles let you view the arcs for a single function either as a caller or callee. You do this by entering the function name in the field and then clicking the appropriate toggle, or **CallerCallee**.

- **List Instrumentation**: displays the instrumentation information for a particular test. See Figure 5-23, page 97.

  Function List toggle shows the functions that are included in the coverage experiment.

Ver allows you to specify the version of the program that was instrumented. The latest version is used by default.

Executable displays the executable associated with the current coverage data. You can switch to a different executable by entering it directly in this field. You can also switch executables by clicking the **Executable List** button, selecting from the list in the dialog box, and clicking **Apply** in the dialog box.

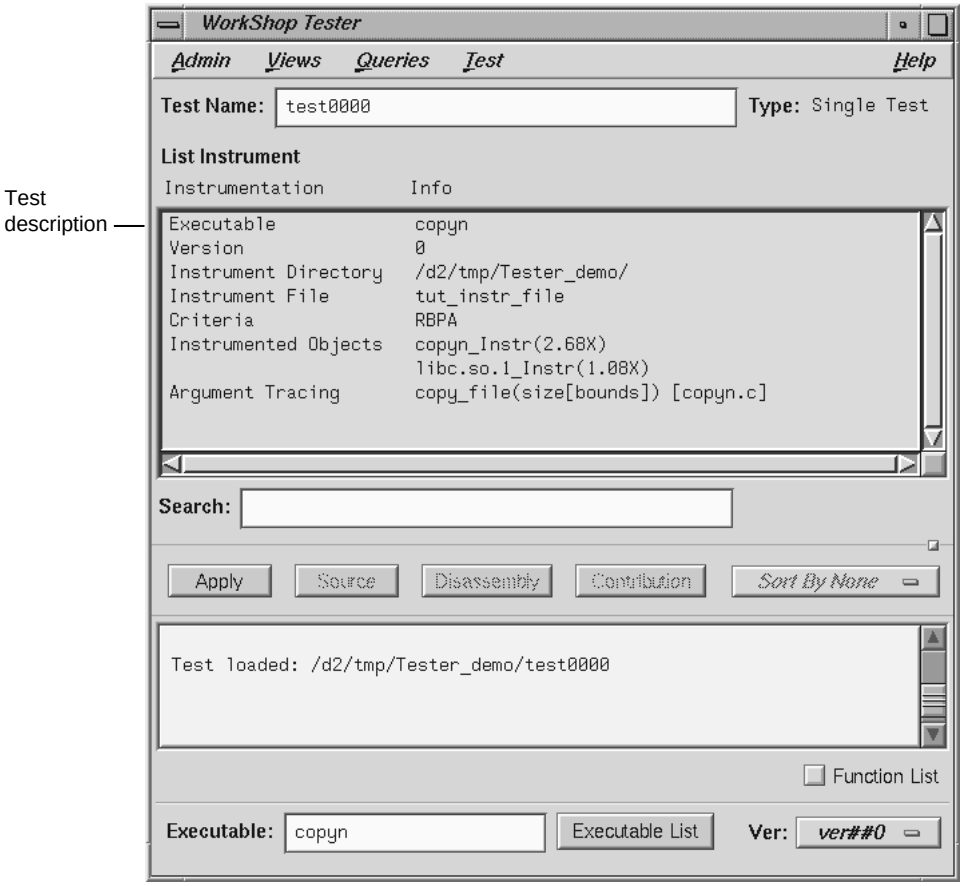

Figure 5-23 List Instrumentation Example

- **List Line Coverage**: lists the coverage for each function for native source lines. Entering a function in the **Func Name** field displays the coverage results limited to that function only in the display area. See Figure 5-24.

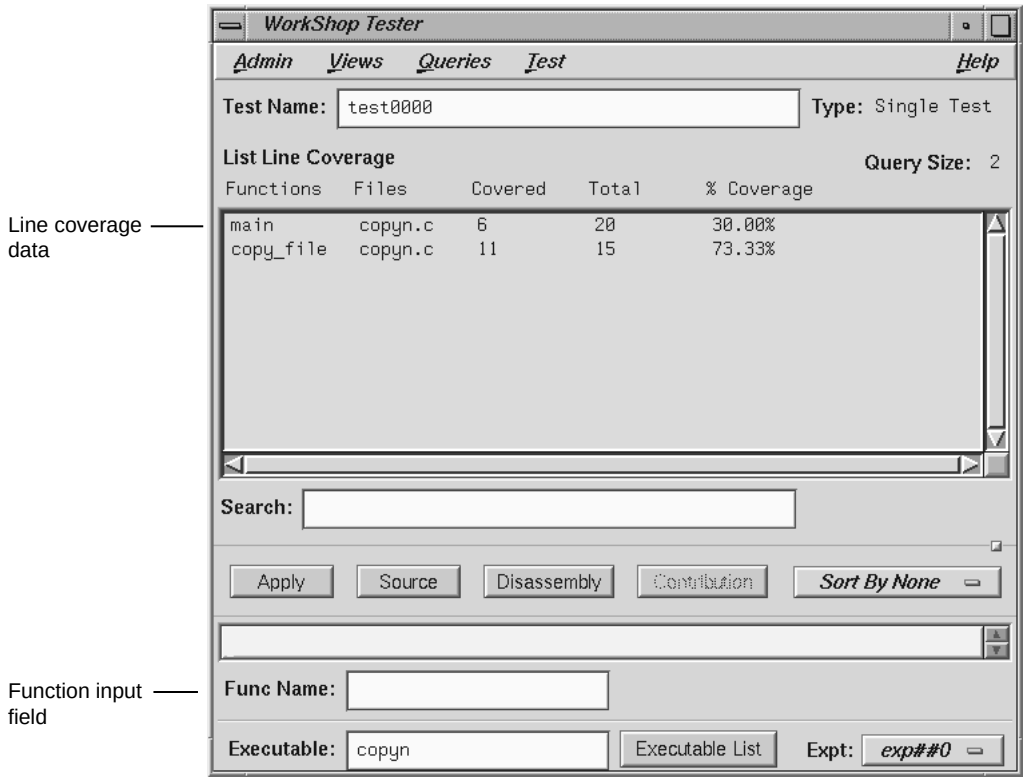

Figure 5-24 "List Line Coverage" Example

- **Describe Test**: describes the details of the test, test set, or test group. When working with test sets and test groups, it is useful to select the **Recursive List** toggle, because it describes the details for all subtests. See Figure 5-25, page 99.

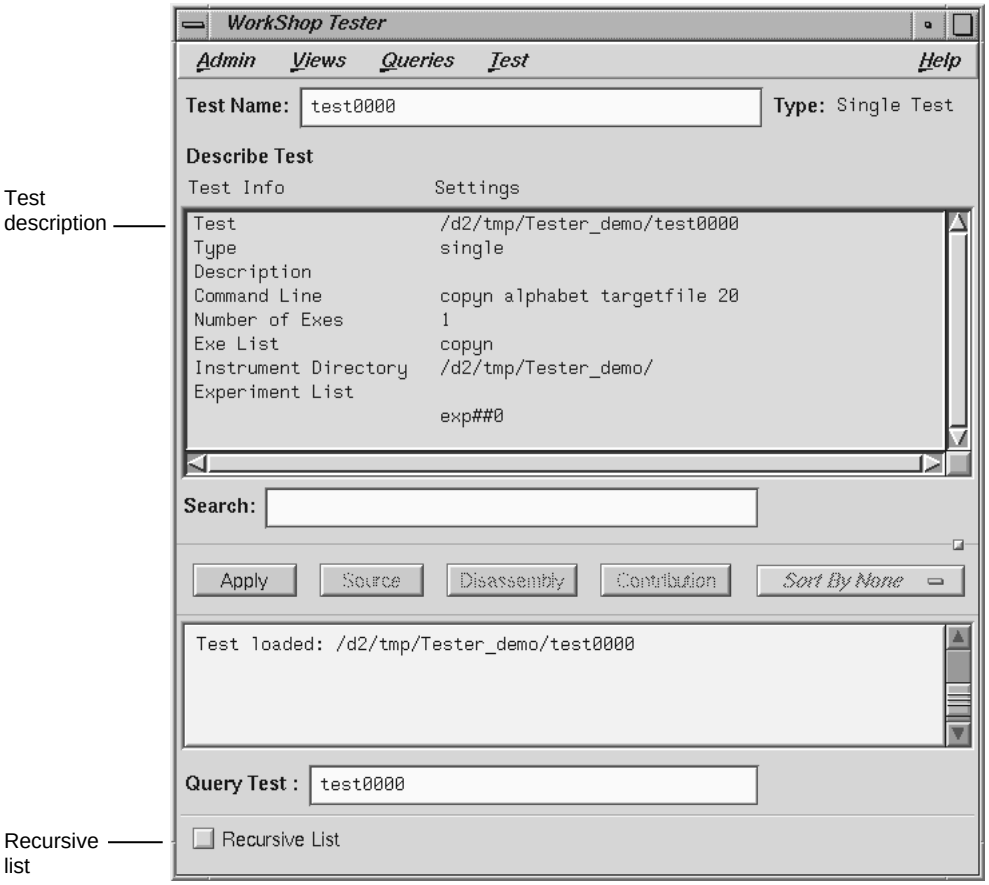

Figure 5-25 Describe Test Example

- **Compare Test**: shows the difference in coverage for the same test applied to different versions of the same program. To perform a comparison, you need to select **Compare Test** from the **Queries** menu, enter experiment directories in the experiment fields, and click **Apply** or press `Return`. The experiments are entered in the form `exp##<n>` if in the same test or in the form `test<nnnn>/exp##<n>` when comparing the results of different tests. See Figure 5-26, page 100.

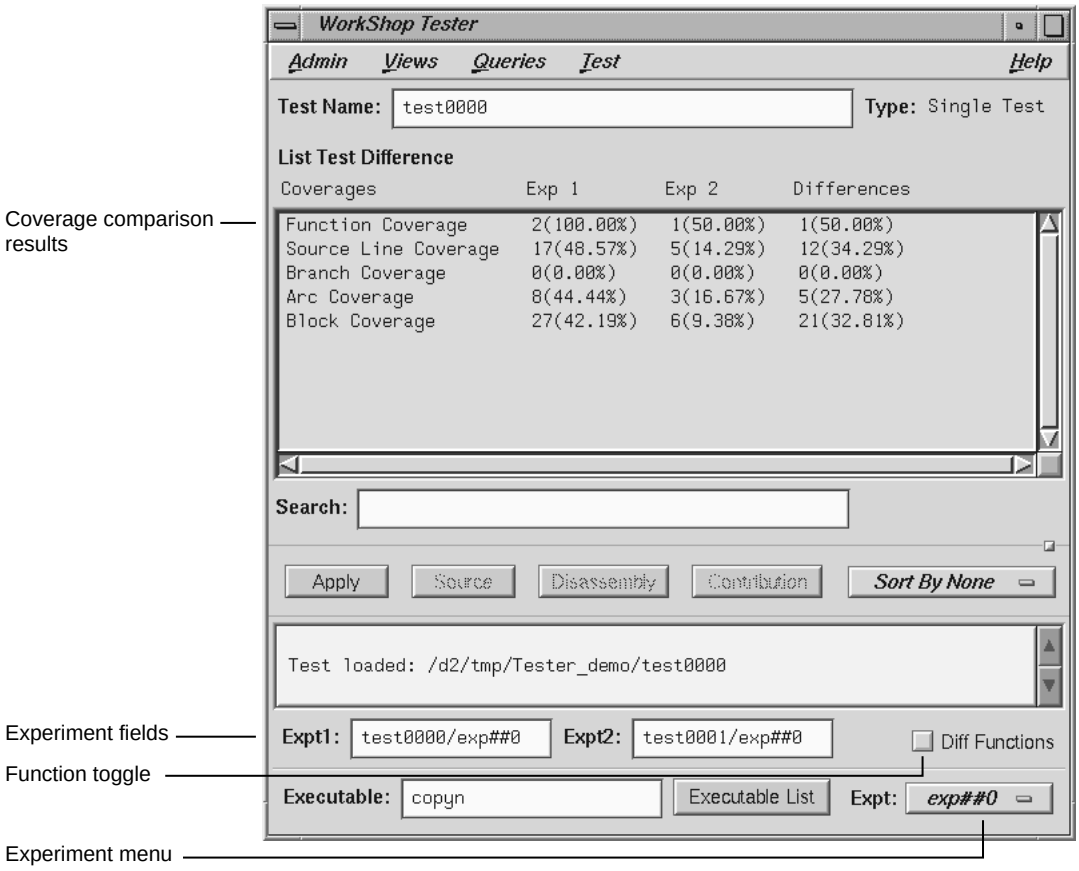

Figure 5-26 Compare Test Example — Coverage Differences

The comparison data displays in the coverage display area. The basic types of coverage display in the **Coverages** column. **Result 1** and **Result 2** display the results of the experiments specified in the **Expt1** and **Expt2** fields, respectively. Results are shown as the counts followed by the coverage percentage in parentheses. The values in the **Result 2** column are subtracted from those in **Result 1** and the differences are shown in the **Differences** column. If you want to view the available experiments, click the **Expt: menu**.

You can also compare the differences in *function coverage* by clicking the **Diff Functions** toggle. Figure 5-27, page 101, shows a typical function difference example.

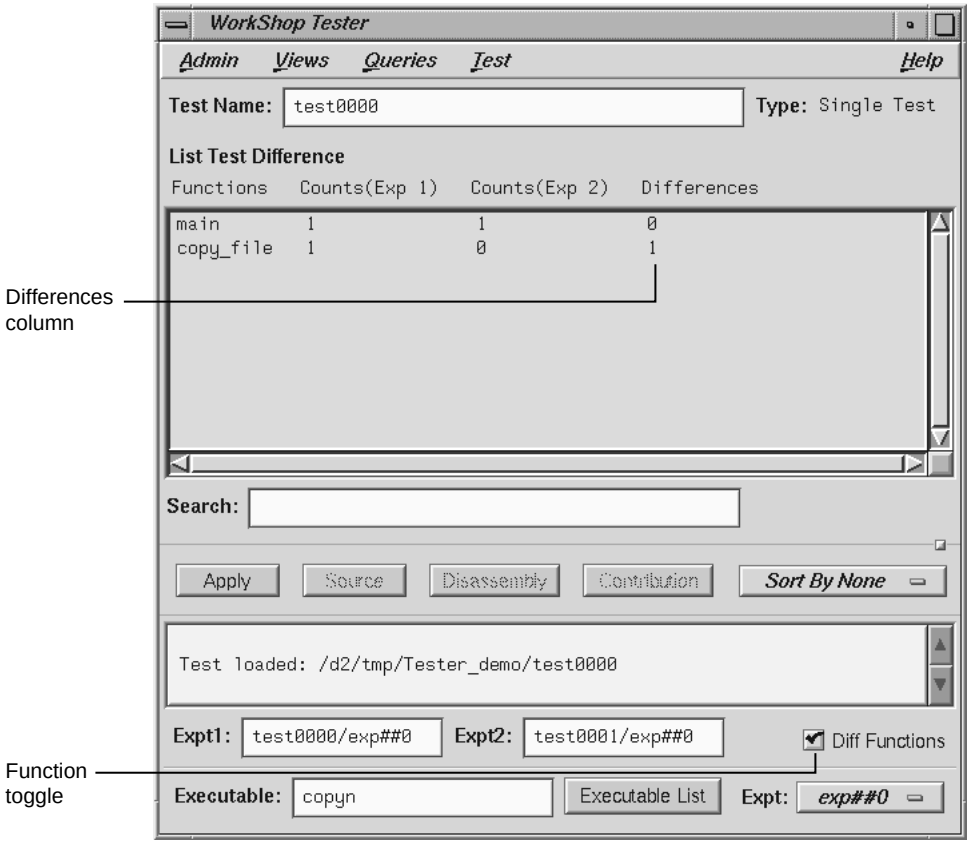

Figure 5-27 Compare Test Example — Function Differences

Admin Menu Operations

The **Admin** menu is shown in Figure 5-28, page 102.

Figure 5-28 Admin Menu

The **Admin** menu has these selections:

- **Save Results**: brings up the standard **File Browser** dialog box so that you can specify a file in which to save the results.
- **Clone Execution View**: displays an **Execution View** window. Use this if you have closed the initial **Execution View** window and need a new one. (You need this window to see the results of **Run Test**.)
- **Set Defaults**: allows you to change the working directory for work on tests in other directories. Also, you can select whether or not to show function arguments. This is useful when distinguishing functions that have the same name but different arguments (for example, C++ constructors and overloaded functions). See Figure 5-29, page 102.

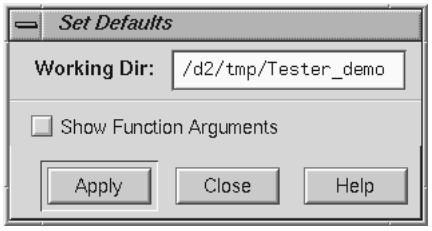

Figure 5-29 "Set Defaults" Dialog Box

- "Launch Tool": the Launch Tool submenu contains commands for launching other WorkShop applications (see Figure 5-30, page 103).

Figure 5-30 Launch Tool Submenu

If any of these tools are not installed on your system, the corresponding menu item will be grayed out.

Exit closes all Tester windows.

cvcov Command Line Examples

This appendix contains several examples of command line usage for `cvcov`. For complete details about using `cvcov`, see the `cvcov`(1) man page and Chapter 3, "Tester Command Line Reference", page 33.

General Test Command Examples

The following examples demonstrate commands that support the creation, inspection, modification, and deletion of tests:

`cattest` describes the test details for a test, test set, or test group.

Example A-1 `cattest` Example

```
% cvcov cattest test0000
Test Info               Settings
------------------------------------------------------------
Test                    /disk2/tutorial/tutorial/test0000
Type                    single
Description
Command Line            copyn alphabet targetfile 20
Number of Exes          1
Exe List                copyn
Instrument Directory    /disk2/tutorial/tutorial/
Experiment List
                          exp##0
                          exp##1
```

Example A-2 `cattest` Example without `-r`

```
% cvcov cattest tut_testset
Test Info                Settings
------------------------------------------------------------
Test                     /disk2/tutorial/tutorial/tut_testset
Type                     set
Description              full coverage testset
Number of Exes           1
Exe List                 copyn
Number of Subtests       9
```

```
Subtest List
                        [0] /disk2/tutorial/tutorial/test0000
                        [1] /disk2/tutorial/tutorial/test0001
                        [2] /disk2/tutorial/tutorial/test0002
                        [3] /disk2/tutorial/tutorial/test0003
                        [4] /disk2/tutorial/tutorial/test0004
                        [5] /disk2/tutorial/tutorial/test0005
                        [6] /disk2/tutorial/tutorial/test0006
                        [7] /disk2/tutorial/tutorial/test0007
                        [8] /disk2/tutorial/tutorial/test0008
Experiment List
                        exp##0
```

Example A-3 `cattest` Example with `-r`

```
% cvcov cattest -r tut_testset

Test Info                Settings
-------------------------------------------------------
Test                    /disk2/tutorial/tutorial/tut_testset
Type                    set
Description             full coverage testset
Number of Exes          1
Exe List                copyn
Number of Subtests      9
Subtest List
                        /disk2/tutorial/tutorial/test0000
                        /disk2/tutorial/tutorial/test0001
                        /disk2/tutorial/tutorial/test0002
                        /disk2/tutorial/tutorial/test0003
                        /disk2/tutorial/tutorial/test0004
                        /disk2/tutorial/tutorial/test0005
                        /disk2/tutorial/tutorial/test0006
                        /disk2/tutorial/tutorial/test0007
                        /disk2/tutorial/tutorial/test0008
Experiment List
                        exp##0
```

`lsinstr` lists the test directories in the current working directory.

Example A-4 `lsinstr` Example

```
% cvcov lsinstr test0000
Instrumentation         Info
-------------------------------------------------------
Executable              copyn
Version                 0
Instrument Directory    /x/tmp/carol/
Instrument File         tut_instr_file
Criteria                RBPA
Instrumented Objects    copyn.pixie(2.57X)
                          libc.so.1_RBP_Instr(1.07X)
```

`mktest` creates a test directory.

Example A-5 Test Description File Examples

```
% cvcov mktest -cmd "copyn tut_instr_file targetfile"
cvcov: Made test directory: /d/Tester/tutorial/test0002

% cvcov cattest test0002
Test Info               Settings
-------------------------------------------------------
Test                    /d/Tester/tutorial/test0002
Type                    single
Description
Command Line            copyn tut_instr_file targetfile
Number of Exes          1
Exe List                copyn
Instrument Directory    /d/Tester/tutorial
Experiment List
```

Coverage Analysis Commands

After the data has been collected from the test experiments, data can be analyzed with special commands for the various types of coverage available.

`lssum` shows the overall coverage based on the user-defined weighted average over function, line, block, branch, and arc coverage.

Example A-6 `lssum` Example

```
% cvcov lssum test0000
Coverages        Covered     Total       % Coverage    Weight
---------------------------------------------------------------
Function         2           2           100.00%       0.400
Source Line      17          35          48.57%        0.200
Branch           0            10           0.00%        0.200
Arc              8           18          44.44%        0.200
Block            19          42          45.24%        0.000
Weighted Sum                             58.60%        1.000
```

`lsfun` lists coverage information for the specified functions in the program that was tested.

Example A-7 `lsfun` Example

```
% cvcov lsfun -pretty -sort function test0000
Functions    Files       Counts
-----------------------------------
copy_file    copyn.c     1
main         copyn.c     1
```

Note: C++ inline functions are not counted as functions.

`lsblock` displays a list of blocks for one or more functions and the count information associated with each block.

Example A-8 `lsblock` Example

```
cvcov lsblock -pat main -pretty test0000
Blocks      Functions    Files       Counts
-----------------------------------------------
13~16       main         copyn.c     1
17~17       main         copyn.c     0
18~18       main         copyn.c     0
19~19       main         copyn.c     0
21~21       main         copyn.c     1
22~22       main         copyn.c     0
23~23       main         copyn.c     0
24~24       main         copyn.c     0
```

```
26~26       main         copyn.c     1
26~27       main         copyn.c     1
27~27       main         copyn.c     1
28~28       main         copyn.c     0
28~28(2)    main         copyn.c     0
28~28(3)    main         copyn.c     0
28~28(4)    main         copyn.c     0
30~30       main         copyn.c     0
31~31       main         copyn.c     0
33~33       main         copyn.c     0
34~34       main         copyn.c     0
36~36       main         copyn.c     0
37~37       main         copyn.c     0
39~39       main         copyn.c     0
41~41       main         copyn.c     0
43~43       main         copyn.c     1
43~43(2)    main         copyn.c     0
43~43(3)    main         copyn.c     1
```

`lsbranch` lists coverage information for branches in the program, including the line number at which the branch occurs.

Example A-9 `lsbranch` Example

```
% cvcov lsbranch -pretty -sort function test0000
Line        Functions    Files       Taken       Not Taken
-------------------------------------------------------------
50          copy_file    copyn.c     1           0
54          copy_file    copyn.c     1           0
57          copy_file    copyn.c     1           0
60          copy_file    copyn.c     1           0
16          main         copyn.c     1           0
21          main         copyn.c     1           0
27          main         copyn.c     1           0
28          main         copyn.c     0           0
28(2)       main         copyn.c     0           0
28(3)       main         copyn.c     0           0
```

`lsarc` shows *arc coverage*, that is, the number of arcs taken out of the total possible arcs.

Example A-10 `lsarc` Example

```
% cvcov lsarc -callee printf -pretty test0001
Callers    Callees     Line       Files       Counts
-----------------------------------------------------
main        printf      17          copyn.c     1
main        printf      18          copyn.c     1
main        printf      22          copyn.c     0
main        printf      23          copyn.c     0
main        printf      30          copyn.c     0
main        printf      33          copyn.c     0
main        printf      36          copyn.c     0
```

`lscall` lists the call graph for the executable with counts for each function.

Example A-11 `lscall` Example

```
% cvcov lscall -pretty test0000
Graph            Counts
---------------------------------
main             1
     copy_file     1
        _open      N/A
        _stat...   N/A
        _creat     N/A
        _malloc... N/A
        _read      N/A
        _write     N/A
     printf...     N/A
     exit...       N/A
     atoi          N/A
```

Lists the coverage for native source lines.

Example A-12 `lsline` Example

```
% cvcov lsline -pretty -pat main test0000
Functions   Files      Covered   Total     % Coverage
-------------------------------------------------------
main        copyn.c    6         20        30.00%
```

`lssource` displays the source annotated with line counts. This option requires the code to be compiled with the `-g` option.

Example A-13 `lssource` Example

```
% cvcov lssource main test0000
Counts  Source
-------------------------------------------------------------
          #include <stdio.h>
          #include <sys/stat.h>
          #include <sys/types.h>
          #include <fcntl.h>

          #define OPEN_ERR         1
          #define NOT_ENOUGH_BYTES 2
          #define SIZE_0           3

          int copy_file();

          main (int argc, char *argv[])
1       {
              int bytes, status;

1           if( argc < 4){
0               printf(''copyn: Insufficient arguments.\n'');
0               printf(''Usage: copyn f1 f2 bytes\n'');
0               exit(1);
              }
1           if( argc > 4 ) {
0               printf(''Error: Too many arguments\n'');
0               printf(''Usage: copyn f1 f2 bytes\n'');
0               exit(1);
              }
1           bytes = atoi(argv[3]);
```

`diff` shows the difference in coverage for different versions of the same program. The following examples show different uses of the `diff` option.

Example A-14 `diff` between Two Tests

```
% cvcov diff test0000/exp##0 test0001/exp##0

Experiment 1:    test0000/exp##0
Experiment 2:    test0001/exp##0
```

```
Coverages            Exp 1       Exp 2        Differences
-------------------------------------------------------------
Function Coverage    2(100.00%)  1(50.00%)    1(50.00%)
Source Line Coverage 17(48.57%)  5(14.29%)    12(34.29%)
Branch Coverage      0(0.00%)    0(0.00%)     0(0.00%)
Arc Coverage         8(44.44%)   3(16.67%)    5(27.78%)
Block Coverage       19(45.24%)  4(9.52%)     15(35.71%)
```

Example A-15 `diff` between Different Instrumentations of the Same Test

```
% cvcov diff test0000/exp##0 test0000/exp##1
Experiment 1:    test0000/exp##0
Experiment 2:    test0000/exp##1

Coverages            Exp 1        Exp 2       Differences
-------------------------------------------------------------
Function Coverage    2(100.00%)    2(100.00%)    0(0.00%)
Source Line Coverage 17(48.57%)    17(47.22%)    0(1.35%)
Branch Coverage      0(0.00%)      0(0.00%)      0(0.00%)
Arc Coverage         8(44.44%)     8(44.44%)     0(0.00%)
Block Coverage       19(45.24%)    19(44.19%)    0(-1.05%)
```

Test Set Command Examples

A test set is a named collection of tests and other test sets. Test sets can be hierarchical. There are several commands used with test sets, including `mktset`, `addtest`, `deltest`, and `optimize`.

`optimize` selects the minimum set of tests that give the same coverage or meet the given coverage criteria as the given set.

Example A-16 Optimizing Test Sets

```
% cvcov optimize -pretty -blocks -branches test00*

Test                Block Coverage    Branch Coverage
-------------------------------------------------------------
test0000            41.54%            0.00%
test0001            7.69%             10.00%
test0002            7.69%             10.00%
test0003            9.23%             20.00%
```

```
test0004            9.23%           20.00%
test0005            6.15%           20.00%
test0006            1.54%           10.00%
Total Coverage      83.08%          90.00%
```

Glossary

anti-leak

See bad free.

arc

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

Array Browser

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

bad free

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

bar graph view

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

basic block

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

boundary overrun

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

boundary underrun

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

breakpoint

See trap (breakpoint) and watchpoint (data-breakpoint).

Browser (Static Analyzer)

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

Build Analyzer

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

Build Manager

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

Build View

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

calipers

See time line.

call graph

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

Call Graph

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

Call Graph View

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

Call Stack View

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

Call Tree View (Static Analyzer version)

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

Call Tree View (Tester version)

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

Call View

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

Class Graph

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

Class Tree View

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

command line (Debugger)

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

cord

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

Cord Analyzer

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

counts

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

coverage

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

CPU-bound

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

CPU time

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

cvcord

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

cvcov

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

cvd

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

cvperf

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

cvspeed

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

cvstatic

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

cvxcov

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

cycle count

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

Debugger

A tool in the ProDev WorkShop toolkit used for analyzing general software problems using a live process. The Debugger lets you stop the process at specific locations in the code by setting breakpoints (referred to as traps) or by clicking the Stop button. At each trap, you can examine data by displaying special windows called views. See also `cvd`.

Disassembly View

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

double free

See bad free.

DSO (dynamic shared object)

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

erroneous free

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

event

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

exclusive performance data

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

Execution View

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

experiment

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

Expression View

A Debugger view that lets you specify one or more expressions to be evaluated whenever the process stops or the callstack context is changed. Expression View lets you save sets of expressions for subsequent reuse, specify the language of the expression (Ada, Fortran 77, Fortran 90, C, or C++), and specify the format for the resulting values.

File Dependency View

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

Fileset Editor

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

fine-grained usage

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

Fix+Continue

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

floating-point exception

A problem that occurs when a program cannot complete a numerical calculation due to division by zero, overflow, underflow, inexact result, or invalid operand. Floating-point exceptions can be captured by the Performance Analyzer and can also be identified in the Array Browser.

freed memory

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

function list

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

GLDebug

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

heap corruption

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

Heap View

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

ideal time

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

inclusive performance data

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

instrumentation

See experiment.

I/O-bound

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

I/O View

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

IRIS IM

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

IRIS IM Analyzer

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

IRIS ViewKit

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

Leak View

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

library search path

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

Main View

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

Malloc Error View

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

malloc debugging library

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

Malloc View

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

MegaDev

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

Memory-bound

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

memory leak

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

Memory View

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

Multiprocess View

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

node

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

Overview window

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

parser mode

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

Path Remapping

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

PC (program counter)

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

Performance Analyzer

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

Performance Panel

A window for setting up Performance Analyzer experiments. The panel displays toggles and fields for specifying data to be captured. As a convenience, you can select performance tasks (such as **Determine bottlenecks...** or **Find memory leaks**) from a menu that specifies the data automatically. See also`cvspeed`(1).

phase

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

pollpoint sampling

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

Process Meter

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

ProDev WorkShop

The package name for the core WorkShop tools.

profile

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

Project View

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

query

The term for a search through a Static Analyzer database to locate elements in your program. Queries are similar to the IRIX `grep`(1) command but provide a more specific search. For example, you can perform a query to find where a method is defined. See also Static Analyze

Register View

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

Results Filter

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

sample trap

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

sampling

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

scanner mode

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

Signal Panel

A dialog box for specifying signals to trap.

Smart Build

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

Source View

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

stack

See Call Stack.

Static Analyzer

A tool in the ProDev WorkShop toolkit used for viewing the structure of a program at different levels and locating where elements of the program are used or defined. The Static Analyzer works by extracting structure and location information from files that you specify and storing the information in a database for subsequent analysis. You can view the analysis as a text list or graphically. See also`cvstatic`(1), **Call Tree View**, **Class Tree View**, **File Dependency View**, and **Text View**.

stop trap

A breakpoint. See also trap.

Data Explorer

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

Syscall Panel

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

test group

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

test set

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

Tester

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

Text View (Static Analyzer version)

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

Text View (Tester version)

A Tester view that displays function coverage information in a report form. See also Tester.

time line

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

tracing

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

trap

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

There are two types of traps: stop traps are used in debugging to halt a process, and sample traps are used in performance analysis to collect data while halting the process only briefly (and continuing execution automatically). See also watchpoint.

Trap Manager

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

uninitialized memory

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

Usage View (Graphical)

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

Usage View (Textual)

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

Variable Browser

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

view

A window that lets you analyze data.

ViewKit

See IRIS ViewKit.

watchpoint

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

working set

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

Working Set View

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

Index