USER'S GUIDE

TCAT C/C++ for Windows

Version 2.1

Test Coverage Analysis Tool
For C and C++ on
Windows 95 and Windows NT

SOFTWARE RESEARCH, INC.

This document property of:

Name:_______________________________

Company:____________________________

Address:_____________________________

____

Phone________________________________

SOFTWARE RESEARCH, INC.

625 Third Street

San Francisco, CA 94107-1997

Tel: (415) 957-1441

Toll Free: (800) 942-SOFT

Fax: (415) 957-0730

E-mail: support@soft.com

http://www.soft.com

(Last Update December 18, 1998)

home/l1/wu/win-tcat/tcat21/tcatwin2.1

Table of Contents

List of Figures

Preface

Congratulations!

By choosing the TestWorks suite of testing tools, you have taken the first step in bringing your application to the highest possible level of quality.

Software testing and quality assurance, while increasingly important in today's competitive marketplace, can dominate your resources and delay your product release. By automating the testing process, you can assure the quality of your product without needlessly depleting your resources.

Software Research, Inc. believes strongly in automated software testing. It is our goal to bring your product as close to flawlessness as possible. Our leading-edge testing techniques and coverage assurance methods are designed to give you the greatest insight into your source code.

TCAT C/C++ for Windows is a quick and easy way to detect weaknesses in your code. Easily accessible click-and-point reports find the segments that need further testing. Digraphs and calltrees visualize the location, allowing you to make immediate improvements to the structure and performance of your software.

TestWorks is the most complete solution available, and the peace of mind it provides our customers is our most valued feature.

Thank you for choosing TestWorks.

Audience

This manual is intended for software testers who are using *TCAT C/C++ for Windows*. You should be familiar with the Microsoft Windows System and your workstation.

Typefaces

Typographical conventions that are used throughout this manual:

boldface — Introduces or emphasizes a term that refers to **TestWorks'** window, its submenus and its options.

italics — Indicates the names of files, directories, pathnames, variables, and attributes. Italics is also used for manual, chapter, and book titles.

"Double Quotation Marks" — Indicates chapter titles and sections. Words with special meanings can also be set apart with double quotation marks the first time they are used.

`courier` — Indicates system output such as error messages, system hints, file output, and *CAPBAK/MSW*'s keysave file language.

`Boldface Courier` — Indicates any command or data input that you are directed to type. For example, prompts and invocation commands are in this text. (**`stw`**, for instance, invokes *TestWorks*.)

CHAPTER 1

TCAT C/C++ for Windows Overview

This chapter is a conceptual introduction to coverage tools, and explains how to use them most advantageously.

1.1 The QA Problem

It is a sad fact of the software engineering world that on average, without coverage analysis tools, only around 50% of source code is actually tested before release. With little more than half of the logic covered, many bugs go unnoticed until after release. Worse still, the actual percentage of logic covered is unknown to SQA management, making any informed decisions impossible.

Questions such as when to stop testing or how much more testing is required are answered not on the basis of data, but on ad hoc comments and sketchy impressions. Software developers are forced to gamble with the quality of the released software and to make plans based on inadequate data.

A related problem is that test case development is done in an inefficient manner; that is, many test cases are redundant. Test suites become cluttered with cases that repeatedly test the same logic, to the exclusion of other cases that would examine previously unexplored logic. Often, testers are unsure of which direction to take, and can waste SQA time devising the wrong tests.

1.2 The Solution

The primary purpose of testing is to ensure the reliability of a software program before it is released to the end user. The software should be thoroughly tested with a variety of input to provide statistically verifiable means of demonstrating reliability. In other words, a suite of test cases should in some way cover all the possible situations in which the program will be used.

It is a worthy goal to imagine every possible use, and to develop and run corresponding test data. However, achieving this goal is extremely complicated and time-consuming. A more realistic goal is to test every part of the program. According to industry studies, achieving this goal yields significant improvement in overall software quality. Coverage analysis improves the quality of your software beyond conventional levels.

1.3 SR's Solution

Software Research, Inc. offers a solution: **TCAT C/C++ for Windows**. This product ensures tests that are more diverse than those chosen by reference to functional specification alone or those based on a programmer's intuition. It ensures that they are as complete as possible by measuring against a range of high-quality test metrics:

- Coverage at the logical branch (or segment) level and the call-graph level, employing the *C1* metric

  You can choose to test a single module, multiple modules, or the entire program using the *C1* metric.

- Coverage at the call-pair level employing the *S1* metric

  After individual modules have been tested, you can test all the interfaces of the system using the *S1* metric.

- Dynamic visualization of test attainment during unit testing and system integration

  This visually demonstrates, in real time, such things as segments and call-pairs hit / not hit.

Below is a **TCAT C/C++ for Windows** flow chart.

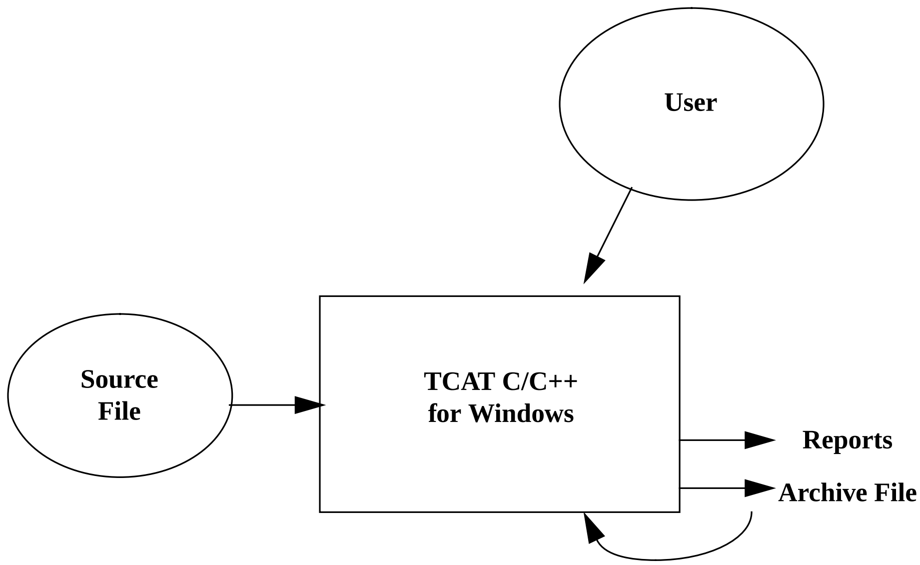


FIGURE 1 TCAT C/C++ for Windows Dependency Chart

1.4 Testing and TCAT C/C++ for Windows

TCAT C/C++ for Windows instruments your program. During instrumentation, **TCAT C/C++ for Windows** inserts function calls (special markers) at every logical branch (segment) in each program module. Instrumentation also creates a reference listing file, which is a version of your program which has logical branch marking comments added to it in a manner similar to the code added to the instrumented version. Extensive logical branch notation and sequence numbers are also listed.

This instrumented program is then compiled and run. By running it, you are exercising logical branches in the program. The more tests in your test suite, the higher the coverage. This test information is then written to a trace file. From the information stored in the trace file, you can generate coverage reports. In general, the reports give the following information:

- Reports included in the current iteration
- A summary of past coverage runs
- Current and cumulative coverage statistics
- A list of logical branches that have been hit

Recommended coverage is >85%. If reports indicate that you have less than this amount, you can identify unexercised logical branches by studying the coverage reports, and looking at the source code associated with the untested functions. When you identify the troubled areas, you can then create new test cases and re-execute the program.

TCAT C/C++ for Windows can help you reach your goal of creating the most extensive test cases possible.

1.5 Software Test Methods

Coverage analysis as implemented through **TCAT C/C++ for Windows** is a powerful testing technique which can save you much money and time, in addition to greatly improving software quality. It is not the only testing technique in existence, and we recommend that you use it along with other techniques.

Testing methods vary from shop to shop, but most successful techniques fall into a few general categories. The most common ones are described below in the sequence they usually occur.

1.5.1 Manual Analysis

Programs are manually inspected for conformance to in-house rules of style, format, and content as well as for correctly producing the anticipated output and results. This process is sometimes called “code inspection,” “structured review,” or “formal inspection.”

1.5.2 Static Analysis

Once a program has passed through manual testing steps, it can be tested more extensively. Automated tools are used to check the design rules applied in a program. Static analysis validates the software allegations about the program's static properties, such as the global properties of its data structures and the application of variable type rules. Such testing can remove 20-30% of the latent software defects in your program. Static analyzers include the following:

- Tools for detecting data element misuse
- Complexity measurement tools, which estimate the difficulty of testing and help identify hard-to-test modules with a statistic
- Conformance measure tools, which flag confusing or inefficient code

1.5.3 Dynamic Analysis

Dynamic analysis tests the dynamic properties of the software under real or simulated operating conditions. The software is executed under controlled circumstances with specific expected results.In this phase, it is important to test as many paths and branches in the program as possible. Doing so ensures that the tests you run have the greatest diversity, hence the best chance of discovering defects.

To obtain statistics on the application under test can be very difficult. Dynamic analysis can uncover 85-90% of the potential remaining software defects. **TCAT C/C++ for Windows** produces data on what has been validated and what has been left out of your testing.

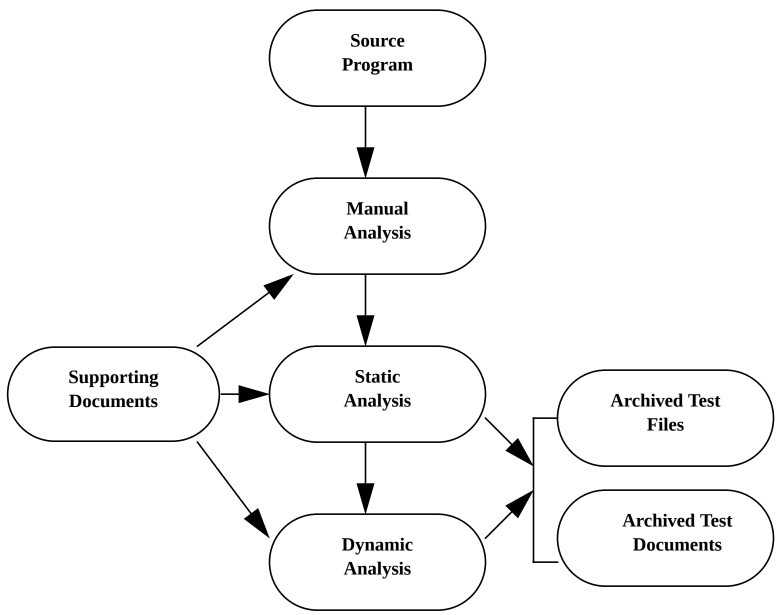


FIGURE 2 Stages in Software Testing

1.6 Single- and Multiple-Module Testing

Another consideration in getting the most out of **TCAT C/C++ for Windows** involves determining the scope of your tests: whether a single program module, multiple modules, or even an entire system should be tested. You can prepare or “instrument” many modules with logical branch markers and run tests on them as a group. **TCAT C/C++ for Windows** keeps track of each module by name.

There are two approaches to multiple-module testing: bottom-up or top-down. Because **TCAT C/C++ for Windows** is able to track many modules simultaneously, it supports either approach. The route you choose depends on your individual needs and testing style.

1.6.1 Buttom-Down

In the bottom-up approach, testing begins at the lowest level in the system hierarchy; that is, modules that invoke no other module. Each bottom-level module is tested individually with special test data. Modules at each subsequent level of the hierarchy are tested using already-tested lower-level modules. The process continues until all modules have been thoroughly exercised. Thus, you can control testing carefully as you progress up the system hierarchy.

1.6.2 Top-Down

In the top-down approach, testing begins at the highest level in the system hierarchy. Sometimes module “stubs”' are used to simulate invoked modules to check the high-level logic of the program. As an alternative to using module stubs, use a complete program with only a few selected modules instrumented. **TCAT C/C++ for Windows** ignores uninstrumented modules as it traces test coverage through the program.

In top-down analysis, the tester is chiefly concerned with the combination of modules to form a larger system. **TCAT C/C++ for Windows** focuses specifically on function calls within the system, so that the tester can verify each interconnection.

1.7 TCAT C/C++ for Windows's Cost Benefits

TCAT C/C++ for Windows will save your organization much time and effort; the economics of coverage analysis are extremely favorable. Here are some ways it can save you money. **TCAT C/C++ for Windows** can save you money in the following ways.

1.7.1 Improved Error Detection

TCAT C/C++ for Windows provides increased error detection. Software Engineering literature indicates that an average error rate is 40 defects per 1,000 lines of code (KLOC). With no coverage analysis, 50% of the code is exercised, leaving the product with 20 defects per KLOC. Assuming a uniform distribution of errors throughout the source code, the simple act of raising the coverage rate can uncover many errors. According to the experience of SR in advanced industrial projects and reports from customers, coverage analysis can eliminate another 75% of the errors.

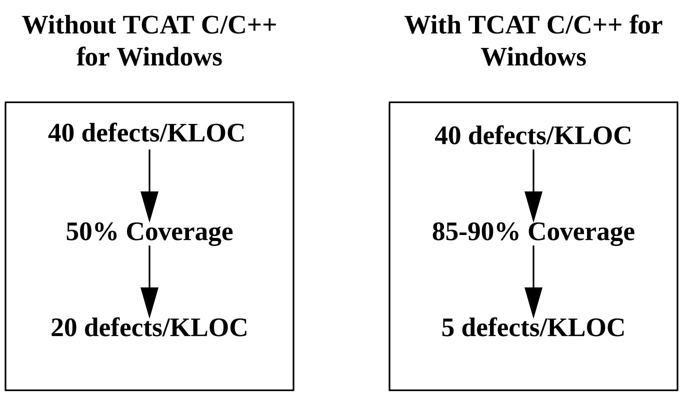


FIGURE 3 Cost Benefit Analysis

The economic value of increased error detection varies from organization to organization. One estimate of the worth of coverage analysis comes from what software consulting firms charge to find and remove errors, a price established in the open market. The software testing industry, sized at $50 million in 1986 by *Fortune* magazine, typically charges $1,000 per error fixed.

Applying this to **TCAT C/C++ for Windows**, you could save $15,000 or more per thousand lines of code. In practical terms, this means that a large project with over 20,000 lines of code might save $300,000.

1.7.2 Earlier Error Detection

Not only are more errors detected with **TCAT C/C++ for Windows**, they are also discovered earlier. The earlier you catch and fix an error, the cheaper. Over and over, managers, vendors and gurus have shown us figures and charts that detail how much less it costs to rectify an early detected defect. The chart below, by Barry Boehm, illustrates this concept.

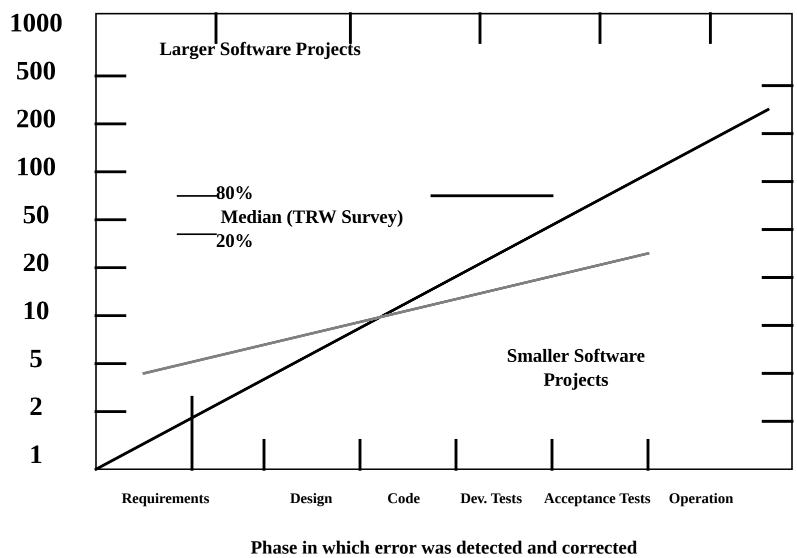


FIGURE 4 Increase in Cost-to-Fix Throughout Life-cycle

Your organization can reduce its cost-to-fix ratio by a factor of ten by using **TCAT C/C++ for Windows** to find errors before system integration. In the diagram, it costs $5,000 to $15,000 to fix errors after they have left the developer. The developer or the Software Quality Engineer (SQE) can identify and fix problems more inexpensively than the beta site or independent testing organization. This is not to say that beta sites or IV&V (independent verification and validation) are not needed; but instead, there is a great cost advantage in letting detailed unit-testing find more errors for less expense.

1.7.3 More Efficient Testing

Using **TCAT C/C++ for Windows**, you can improve test case development. In general, the tool can be used to identify previously untested features. This information can direct the addition of new test cases.

For example, a software test engineer from a super-minicomputer manufacturer used **TCAT C/C++ for Windows** to reduce the time to test by a factor of eight. As detailed in a technical article available from SR, the engineer was in charge of testing a C compiler and used **TCAT C/C++ for Windows** to identify the features missed by commercially-available test suites. The engineer specified the language elements that were not tested to a software engineer, who completed the test suite. Overall, the compiler was fully tested in six weeks rather than the expected one year.

1.7.4 Minimal Test Set

TCAT C/C++ for Windows can be used to develop the minimal covering test suite for a system. It is useful for a tester to have the smallest test suite that exercises all the logic of a system, since test sets require much time and many resources to execute.

We recommend the use of *SMARTS*, *CAPBAK*, and *CBDIFF* (from our *Regression/MSW* tool suite) to automate test suite execution, evaluation, and analysis steps. These tools can significantly reduce the cost of test suite execution and analysis. **TCAT C/C++ for Windows** can be used to identify and eliminate redundant test cases. With the coverage reports described in this manual, it is possible to determine how much each new test case adds to the total coverage of a test suite.

If a new test adds less than a specified amount to the overall coverage (e.g. 5%) it might be reasonable to discard it. Having done so, the tester ends up with more efficient, easier-to-run test suite.

1.7.5 Assessment of Progress

Coverage analysis with **TCAT C/C++ for Windows** can be valuable to important SQA decisions, such as when to ship a product or how much further product testing is needed. A coverage value of C1 > 85% has been the traditional threshold for proper coverage. Generally, one should stop improving test coverage when the marginal cost of adding a new test is greater than the cost to visually and rigorously inspect the associated code passage. Other considerations you can weigh are the added test cost and the risk of defects.

Coverage analysis data are important for reliability modeling and predicting error rates. By tracking error rates and number of errors discovered as a function of overall test effort, it is possible to predict eventual latent defect rates. We encourage SQA managers to keep careful records of errors found and corresponding coverage values.

CHAPTER 2

Installation

This chapter describes the system requirements and the step-by-step installation procedure for TCAT C/C++

2.1 System Requirements

Your computer system must have the following hardware configuration to install and run **TCAT C/C++**.

- Windows 95, or NT4.0.
- 486 microprocessor or better
- 20 MB free disk space.
- 16+ MB RAM recommended

Microsoft Visual C++ must be installed.

2.2 Installation Procedure

These are instructions for installing **TCAT C/C++**.

1. Insert the **CD Disk** in your CDROM drive (these instructions assume D:).
2. **Activate setup.exe. :**

   In Windows 95/NT:

   a. Using either the **My Computer** icon (on the desktop) or **Windows Explorer** (on the **Start** menu, **Programs** submenu), display the contents of the CDROM drive. The **TCAT C/C++ setup.exe** is in ***Coverage* --> *Tcat21*** directory.

   b. Double-click **setup.exe**.

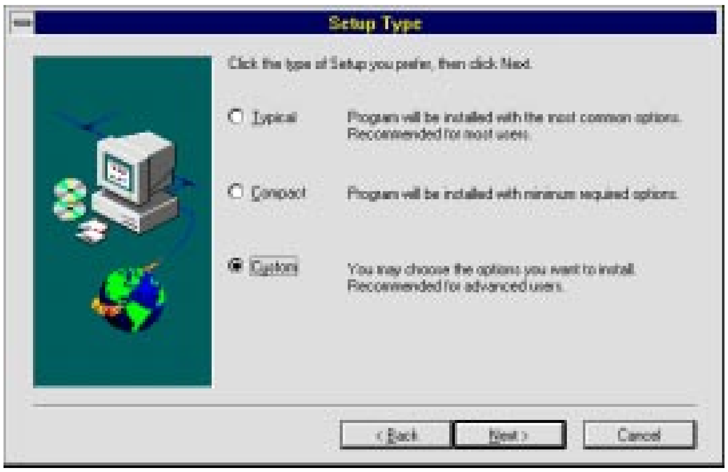
setup.exe presents you with a series of dialog boxes, beginning with the **Welcome** box shown below. Each box is a step in the installation process, and when you are satisfied with the options offered in a box you should click **Next** to go on to the next step.

3. Click **Next** in the **Welcome** box.

   The Choose Destination dialog box asks you where you would like to store the executables and the supporting files for **TCAT C/C++**.

4. To select a path, do one of the following:
   - Click on **Next** if you want to use the **Path** indicated and to continue the installation.
   - Edit the default path to your own path, then click **Next** to continue the installation.
   - Click **Cancel** to end the installation.

After selecting **Next**, the **Setup Type** dialog box pops up and asks you what kind of installation you prefer. It is highly recommended that you select **Custom** installation, which allows you to install the **FrameReader** software that allows you to read the online help that accompanies **TCAT C/C++ for Windows**. (Be aware that the **FrameReader** software will occupy approximately 9 MB of your computer's memory.)



5. In the **Setup Type** dialog box, do one of the following:

- Click **Next** if the Setup Type is the one you prefer.
- Click a different Setup Type, then click **Next** to continue the installation.
- Click **Back** to review or change previous dialog box queries.
- Click **Cancel** to end installation.

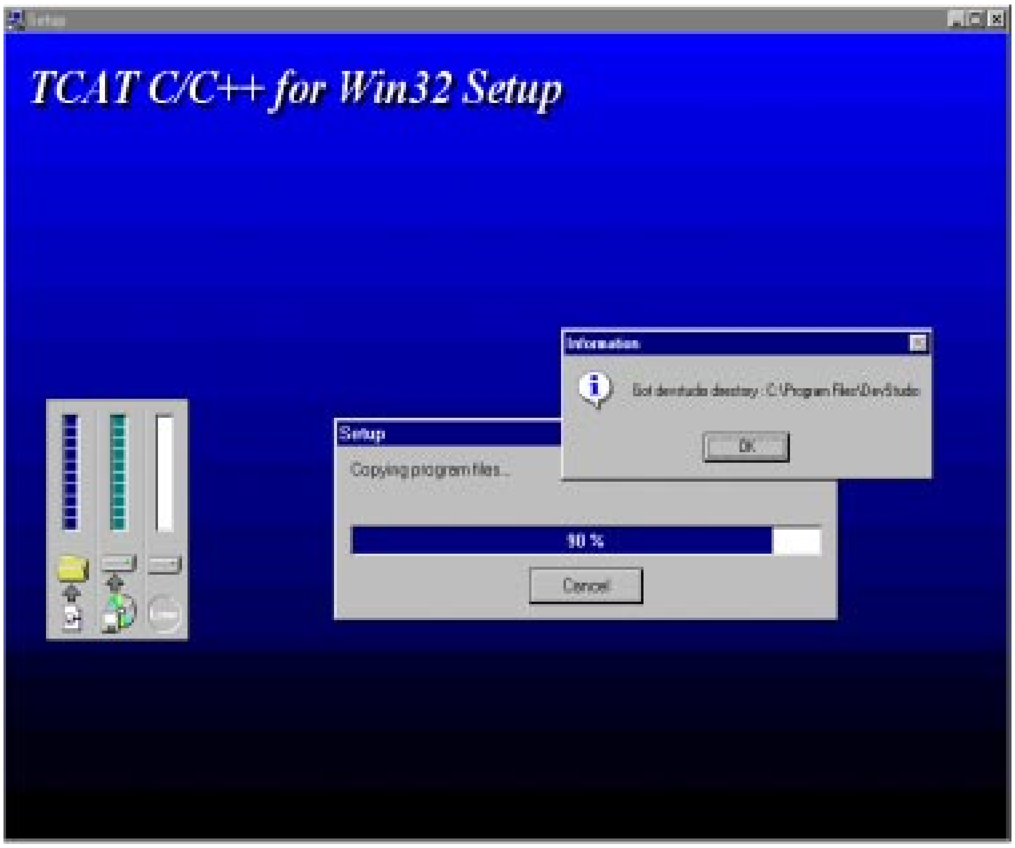
After selecting **Next**, the **Select Components** dialog box pops up in Windows NT and Windows 3.1x, but not in Windows 95. The dialog box asks you to choose the program group name where you would like the program icons to appear.

6. Select the components that you want copied.

During copying, a bar gauge names the files being copied.

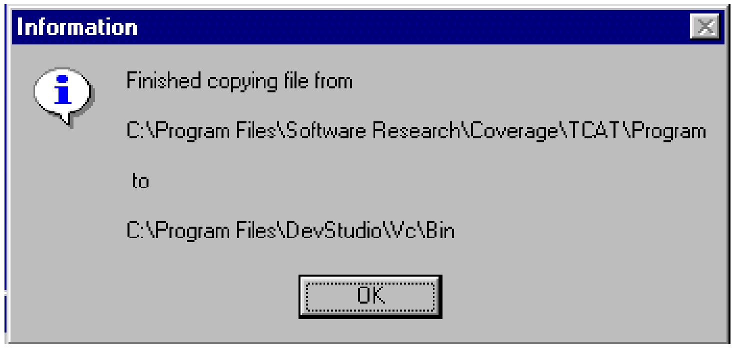
A *C:\Program Files\Software Research\Coverage\TCAT* directory or the path you indicated is created. **TCAT C/C++** automatically stores your files to this directory unless you selected otherwise.

7. The installation verifies where **MS Visual C++** is installed on your machine.



Click **OK** to continue.

During the installation, installation script will copy the ***cl.exe*** file to your **MS DevStudio** in the path specify in the window below and rename its original ***cl.exe*** to ***mscl.exe***.



If you had installed our **TCAT version 2.1** once before, you will get the following window.

Click **OK** to complete the installation.

The installation script also creates a program group where **TCAT C/C++** and its utilities are installed:

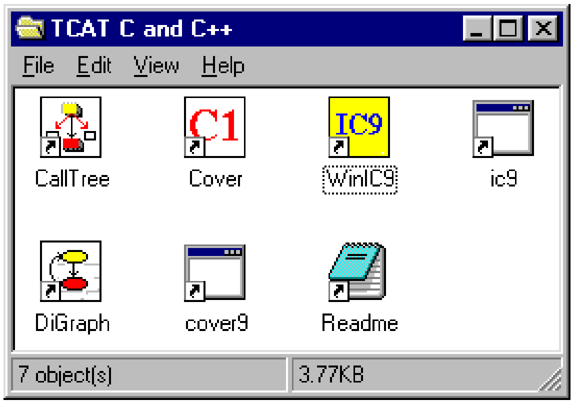


FIGURE 5 Program Group for TCAT C/C++ for Windows

8. When the installation is completed, include the *Coverage* pathname in your system environment variable.
9. To uninstall, use the following:

   In Windows 95 or Windows NT4.0:

   a. Double-click the **Add/Remove Programs** icon in the Control Panel.
   b. Select the **TCAT C and C++ for Win32** option.
   c. Click the **Remove** button.

2.3 File List

The following files are written to your computer during the installation. The locations for these files are given for installation to a directory called *C:\Program Files\Software Research\Coverage\Tcat\Program*.

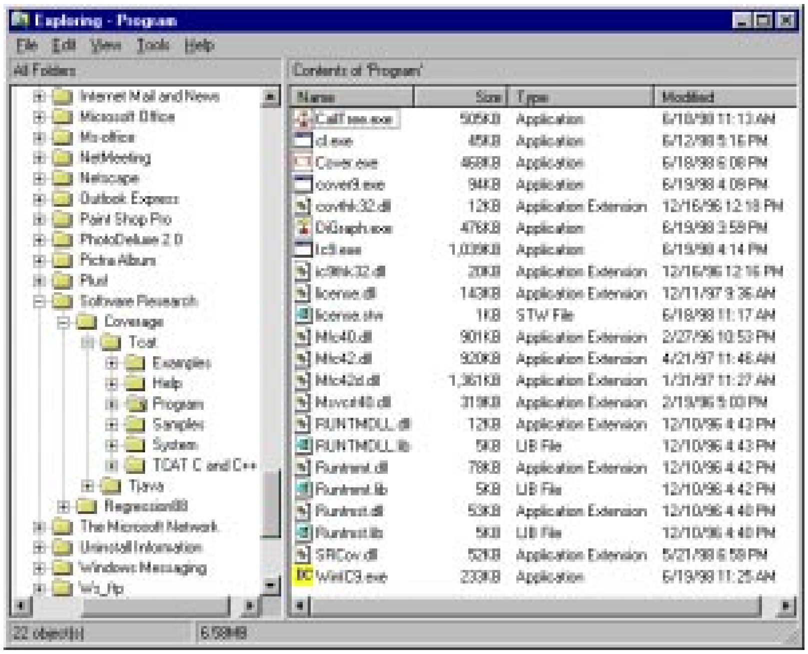


FIGURE 6 Files for TCAT C/C++ in Windows 95/NT

CHAPTER 3

Quick Start

This chapter explains getting started with TCAT C/C++ for Windows using a demonstration test case. It then describes the main features of the product.

3.1 Getting Acquainted with TCAT C/C++ for Windows

This section will familiarize you with the main activities involved in using **TCAT C/C++**, including instrumenting, compiling, linking and running the target program, and finally, looking at resulting coverage reports, calltree graphs and digraphs.

The program used to illustrate the operation of **TCAT C/C++** in Windows is *Scribble*, which you will prepare and instrument as a test application. You can then exercise various logical branches or segments of *Scribble*, creating trace files from which the coverage reports are generated. It is recommended that you complete the *Scribble* example before continuing.

If you are using TCAT C++ for the first time, you will benefit most if you refer to chapters 4 through 7 for in-depth operational instructions and detailed explanation of functionality. If you are an intermediate user, you'll only have to refer to those menu definitions which need further explanation.

3.1.1 Step 1 - Preparing and Instrumenting Scribble

Scribble employs many features of Microsoft Foundation Classes (MFC). There are several versions of **Scribble**, which become increasingly complex in each chapter. MVC++ 5.0 has eight chapters; The present example uses Chapter 8.

This demonstration includes the following steps:

1. Preparing the example application, **Scribble**, for instrumentation.
2. Instrumenting **Scribble**.
3. Building an executable file, ***Scribble.exe***.
4. Testing **Scribble**.
5. Displaying tabular and graphical reports on the test of **Scribble**.

There are two methods to instrument Scribble by either using options from the **TCAT C/C++** Integrated with **MS-Visual C++ v5.0** window or by using the **TCAT C/C++** Program Group window.

3.1.1.1 Using the TCAT C/C++ Integrated with MS-VC++ v5.0 Window

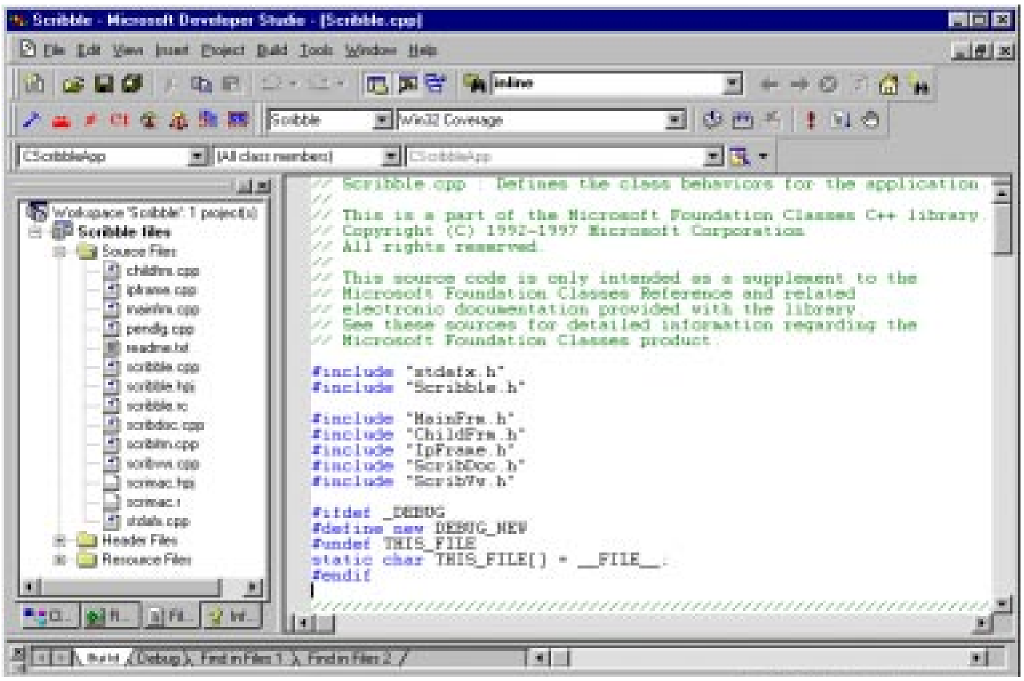


FIGURE 7 TCAT C/C++ Integrated with MS-Visual C++ v5.0 Main Window

1. Select **File|Open Workspace,** then select the "**Scribble.dsw**" file from the Samples directory.

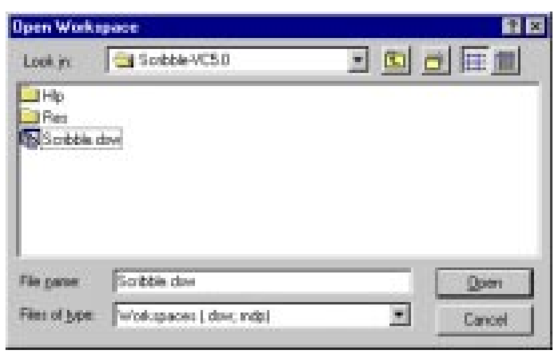


FIGURE 8 Open Workspace Dialog Box

2. From **Build** pull-down menu select **Configuration**, then click the **Add** button and type in "**Coverage**" as a new configuration name**.**
3. Select **Project | Settings,** then select **Win32 Coverage in the box of Setting For:.**
4. From **Project** select **Setting.**
   - Click on the **Scribble** project name, then click on the General tab menu, and type in "Coverage" to both the Output files and Intermediate files option.
   - Click on the **C/C++** tab menu, then select the **Precompiled Headers**, and select the **Not using precompiled headers** options.
   - Click on the "**stdafx.cpp**" file form **Scribble**, then select the **Precompiled Headers** and select **Not using precompiled headers** options.

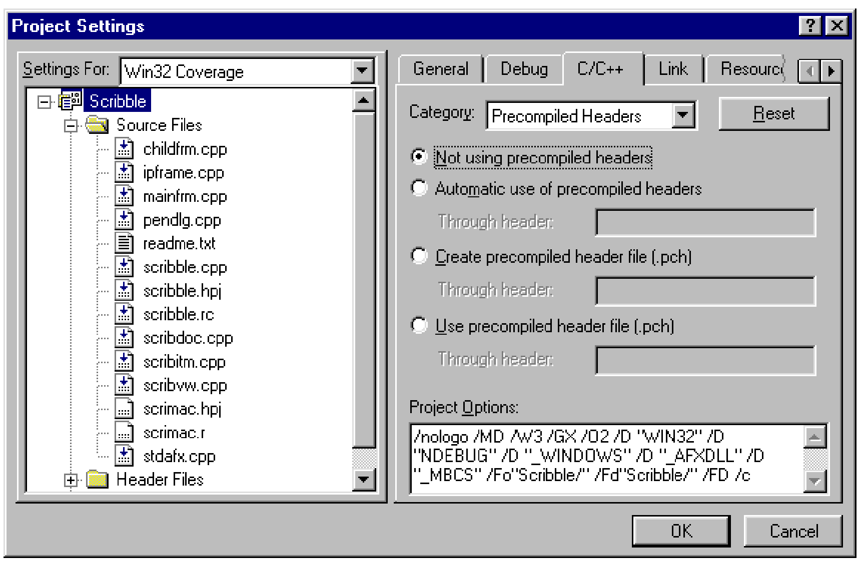


FIGURE 9 Project Setting Dialog Box

5. From **Tools** pull-down menu select **Customize**, then click on the **Add-Ins AND Macro Files** tab menu, and select **SRCov Developer Studio Add-in** option.

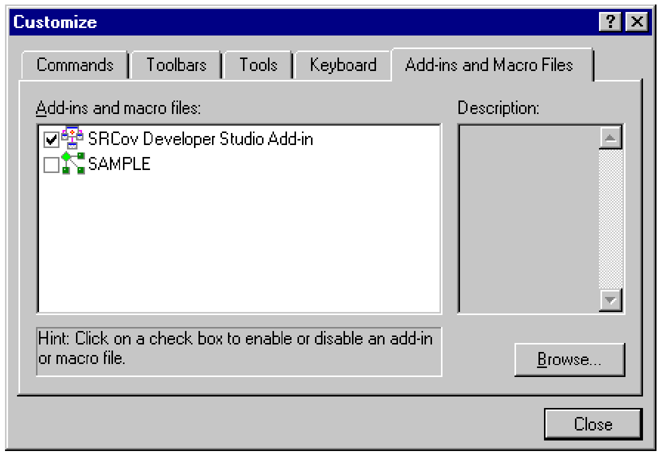


FIGURE 10 Customize Option Dialog Box

The options available from the Tool Bar are the frequently used **TCAT C/C++ for Windows** features.

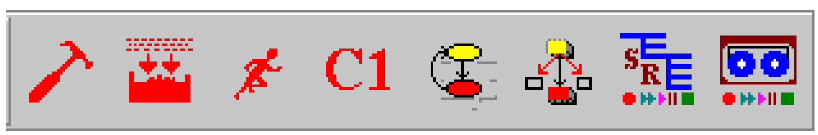


FIGURE 11 Tool Bar

Configure TCAT	Selects among modes of instrumentation.
Build Instrumented App.	Instruments an application.
Run Instrumented App.	Runs the instrumented application.
Analyze Cover	Analyze the coverage achieved from tests.
Run DiGraph	Digraph display for the selected object.
Run Calltree	CallTree display for the selected object.
Run SMARTS	Organizes and executes a collection of tests.
Run CAPBAK	Captures and plays back tool.

6. Click on the **Configure TCAT Option** button.
   - Click on the **Instrumentor Options** tab menu, then select the **C1** and **S1** options.
   - Click on the **Runtime Selection** tab menu, then select the "**RUNTMDLL.lib**" (located in the *Program* directory) file.

3.1.1.2 Instrumenting Scribble

Click on the **Build Instrumented App** button.

The instrumented object files will be placed in the Coverage (debug or release directory if you choose) directory.

3.1.1.3 Executing the Instrumented Scribble

Click on the **Run Instrumented App** button, then test-drive the instrumented **Scribble** to create a trace file.

3.1.1.4 Using the TCAT C/C++ Program Group Window

Setup using Microsoft Visual C++

In **Microsoft Visual C++ v5.0:**

1. Select **File | Open Workspace**, select **Scribble.dsw** (located in the Samples\Scribble directory) as the project.
2. Select **Insert|Files into Project...** and add **RUNTMDLL.lib** (located in the Program directory) to the project.
3. Select **Build|Build Scribble.exe**.

In **Microsoft Visual C++ v4.x:**

1. Select **File | Open Workspace**, select **Scribble.mdp** (located in the Samples\Scribble directory) as the project.
2. Select **Insert|Files into Project...** and add **RUNTMDLL.lib** (located in the Program directory) to the project.
3. Select **Build|Build Scribble.exe**.

3.1.1.5 Instrument Using WinIC9

WinIC9 instruments the application under test so that any tests can produce trace files.

To instrument the example application:

1. Start up **WinIC9** from the TCAT C/C++ program group.

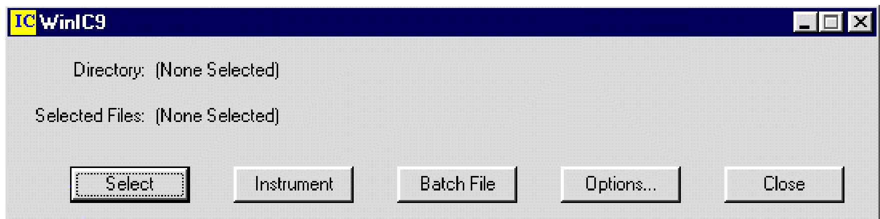


FIGURE 12 WinIC9 Window

2. Select *Scribble.cpp* using the **Select** button. Note that more than one file can be selected and instrumented, and that instrumenting multiple files will result in a more thorough coverage report.

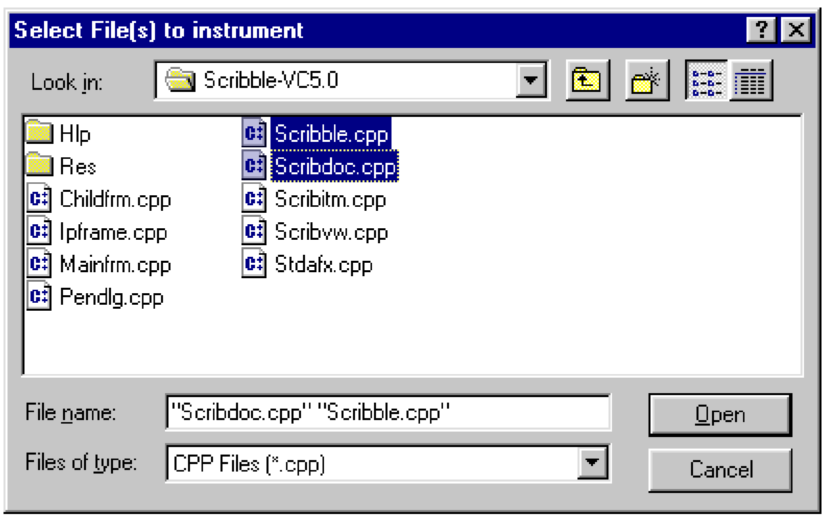


FIGURE 13 Select File(s) to Instrument

Note: More than one file can be selected and instrumented, and instrumenting multiple files results in more thorough coverage.

3. Select **Options** button.

Setting **Compiler Options** for the instrumenter. The TCAT instrumenter invokes the native compiler after completing its processing steps. To instrument a program correctly the compiler options need to be set correctly.

The compiler options very with your application and they can be copied directly from Visual C++ settings. To find the compiler options you need select **Setting** for the project. Then select the appropriate **Project Settings**. Select **C/C++**. The Options that are needed can be found in the field **Project Options**.

One example compiler options setting is listed below.

Scribble Debug Version compiler options:

/nologo /MDd /W3 /Gm /GX /Zi /Od /DWIN32 /D_DEBUG /D_WINDOWS / D_AFXDLL/D_MBCS/Fo". \ Debug/"/Fd"./Debug/"/FD/c

Scribble Release Version compiler options:

/nologo /MD/W3/GX/O2/DWIN32 /NDEBUG/D_WINDOWS/D_AFXDLL/ D_MBCS/Fo". \ Release/"/Fd"./Release/"/FD/c

4. Select **Instrument**. A copyright box pops up before the instrumentation of each file. Click **OK** to proceed.
5. During instrumentation, a command-line window displays messages and warnings. When instrumentation of a file is complete, a prompt appears. Type *exit* to proceed.
6. Select **Exit** from the **WinIC9** window.

The instrumentor has parsed the application's source code, looking for logical branches or segments and inserting markers (function calls).

Instrumenting **Scribble** will not change its functionality. When compiled, linked and executed, the instrumented application will behave as it normally does, except that it will write coverage data to a trace file.

3.1.1.6 Link Using Microsoft Visual C++

In **Microsoft Visual C++**:

1. Build **Scribble.exe**.

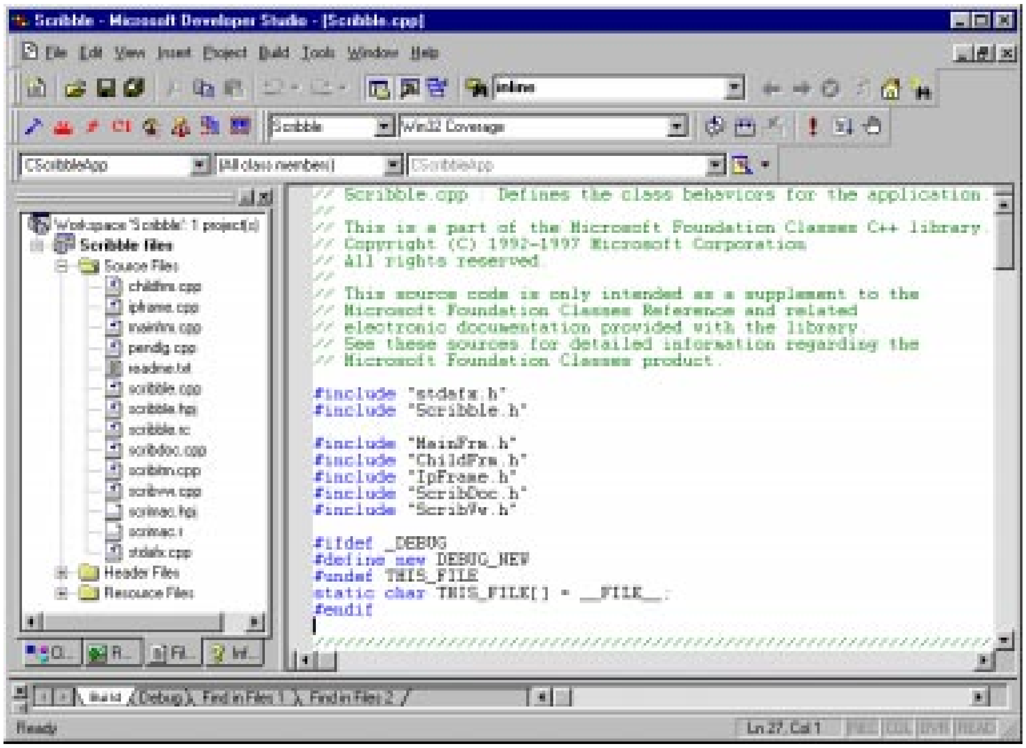


FIGURE 14 TCAT C/C++ Integrated with MS-Visual C++ v5.0 Main Window

The preceding steps create an instrumented executable file for **Scribble**, which when executed will create a trace file.

Instrumenting Scribble.cpp produces the following files in the *Scribble* directory:

- *SCRIBBLE.i* — the instrumented version of the source file

  This file is updated during the instrumentation process.

- *SCRIBBLE.dg* — a Directed Graph Listing file

  Each instrumented file should have its own *.dg* file.

- *SCRIBBLE.cg* — a Calltree Graph Listing file

  Each instrumented file should have its own *.cg* file.

- *SCRIBBLE.mdf* — a Module Definition file

  This file contains information about segments and callpairs in all the processed files.

- *SCRIBBLE.obj* — the instrumented object file

3.1.2 Step 2 - Executing the Instrumented Application

1. Execute **Scribble** from **MSVC++**.
2. Testdrive **Scribble**, as shown in Figure 15.
3. To exit **Scribble**, select **Exit** from the **File** menu.

The trace file created by this "test," *Trace.trc*, resides in the *tcat_db* directory hierarchy in the Scribble directory.

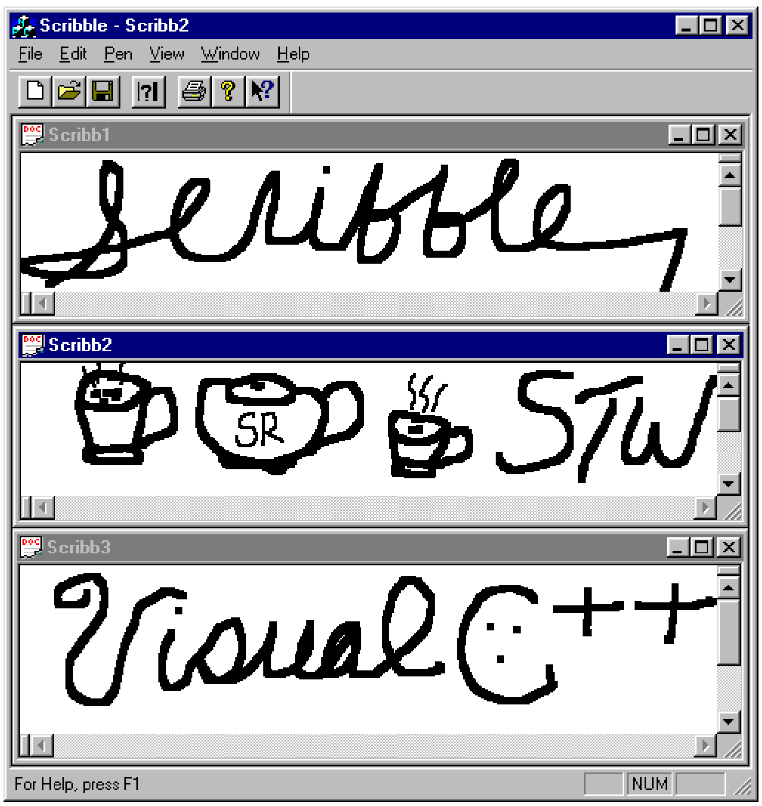


FIGURE 15 Testing Scribble

3.1.3 Step 3 - Viewing Coverage Reports Using Cover

To view a coverage report of the trace file created by the execution of the instrumented version of **Scribble**:

1. Start up **Cover**.
2. From the **File** menu, select **Open**.
3. In the **Open** dialogue**,** click on the filename ***Trace.trc*** from the *tcat_db\Scribble* directory created during instrumentation. The dialog box then asks for an archive file; ignore this request by clicking the **Cancel** button. A coverage report of the test of Scribble appears.

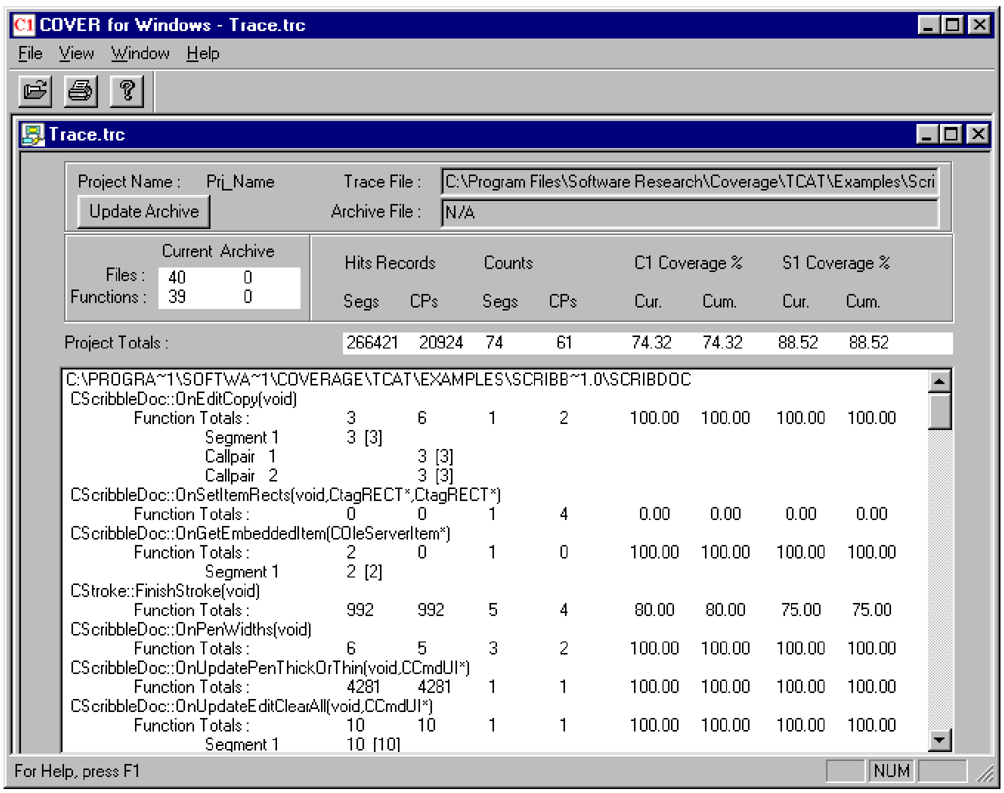


FIGURE 16 Coverage Report on Scribble, with One Function Expanded to Show Segments

Cover displays trace and coverage information on your development project in a treelike list. You can click on a branch of the list to expand it and show its content, and also to contract it. The several fields in the report have the following meanings:

Hits The number of times the segment and call pair were executed during the test

Count The number of segments and call pairs within the function

C1 The percentage of branch coverage for each function

S1 The percentage of call pair coverage for the function

For detailed information about **Cover**, see Chapter 5.

3.1.4 Viewing the Source Code Associated with Cover

You can view the source code associated with any segment numbers, or callpair numbers of the function in a coverage report by clicking on the segment numbers or callpair numbers. For example, click on a segment number. The code is displayed in a separate window with the calling statement highlighted in red.

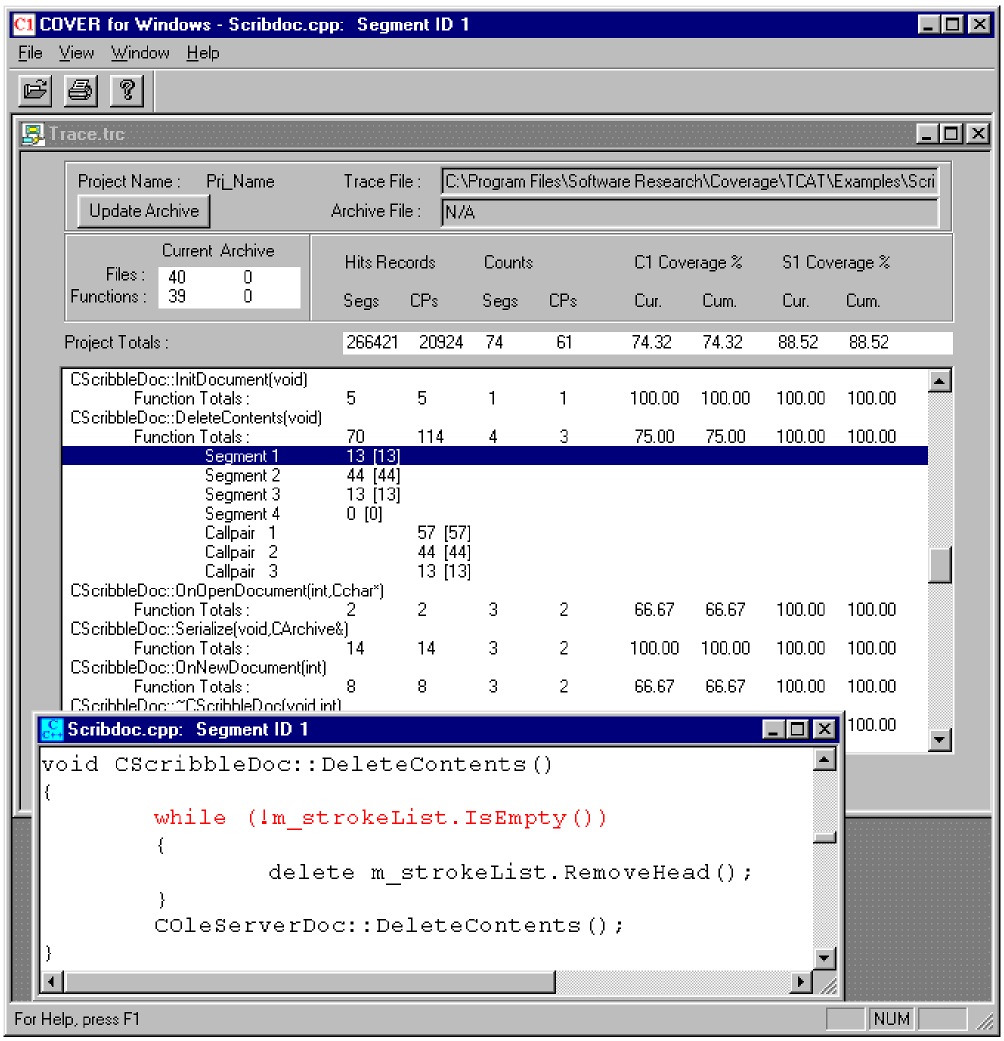


FIGURE 17 Source Code Displayed from Coverage Report

3.1.5 Step 4 - Viewing Directed Graphs with DiGraph

To view a directed graph (digraph) of possible program flows of a function:

1. Open up **DiGraph**.
2. Using the **File** menu, select **Open**.
3. You are prompted for the name of the directed graph to view. Find the *Scribble.dg* file under the *d_graph* directory.
4. The next prompt asks for the name of the database file. Select the *Scribble.mdf* file in the *tcat_db\Scribble* directory.

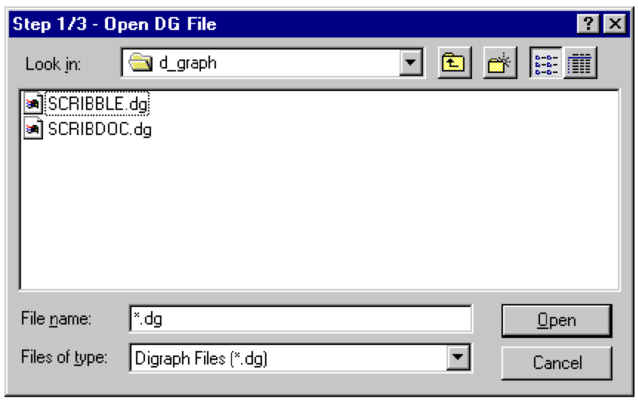


FIGURE 18 WinDiGraph Open Dialog Box

5. A window pops up listing the available functions (Figure 19). For this example, select **CScribbleDoc::DeleteContents[void]**.

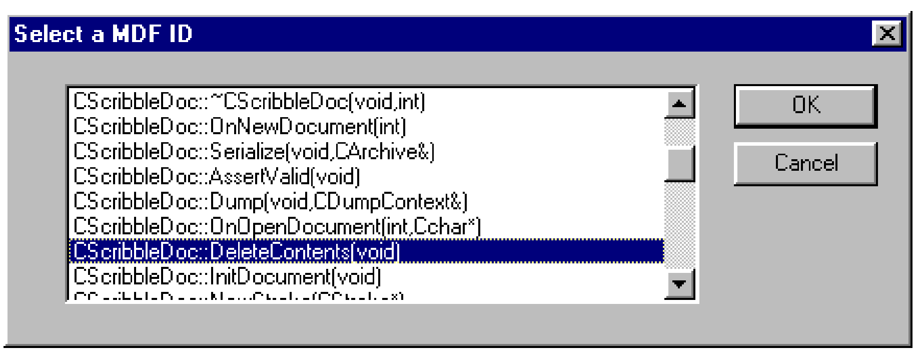


FIGURE 19 Select MDF ID Box

A directed graph depicting possible program flows of the function **CScribbleDoc::DeleteContents[void]** appears.

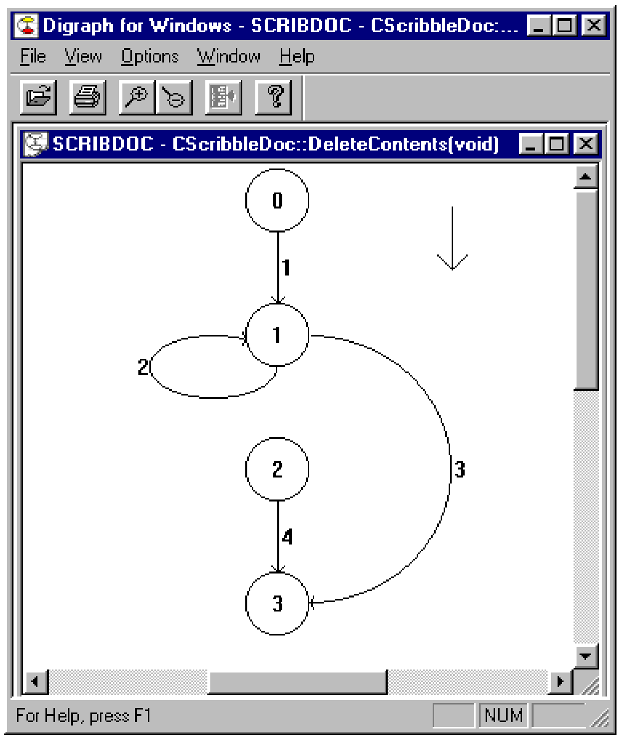


FIGURE 20 Directed Graph of Scribble

The digraph shows the set of conditions and paths that make up a function. The next step shows how to look at the code that the digraph displays as numbered segments.

3.1.6 Step 5 - Viewing Source Code from a Digraph

To view the source code represented by a particular segment of the function **CScribbleDoc::DeleteContents[void]** :

By clicking near the number associated with an edge and selecting the **View Source** button, you can call up and view the associated source code.

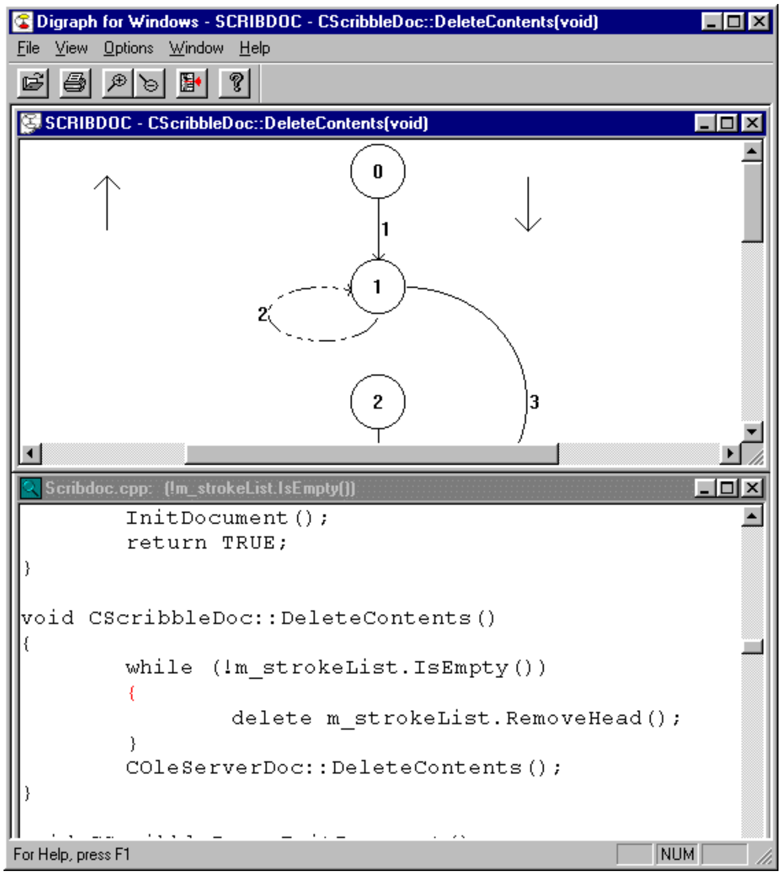


FIGURE 21 Viewing Associated Source Code from Digraph

The source code associated with Segment 2 appears in a new window. In this figure, the windows showing the digraph and the source code have been tiled.

3.1.7 Step 6 - Viewing a Calltree

To view a calltree of **Scribble**:

1. Start up **CallTree**.
2. Using the **File** menu, select **Open**.
3. You are prompted for the name of the calltree to view. Find *Scribble.cg* under the *c_graph* directory.
4. You are prompted for the name of the database file. Find the *Scribble.mdf* file under the *tcat_db* directory.
5. A **Select Function** list box appears. Select the **CScribbleDoc::DeleteContents[void]** function.

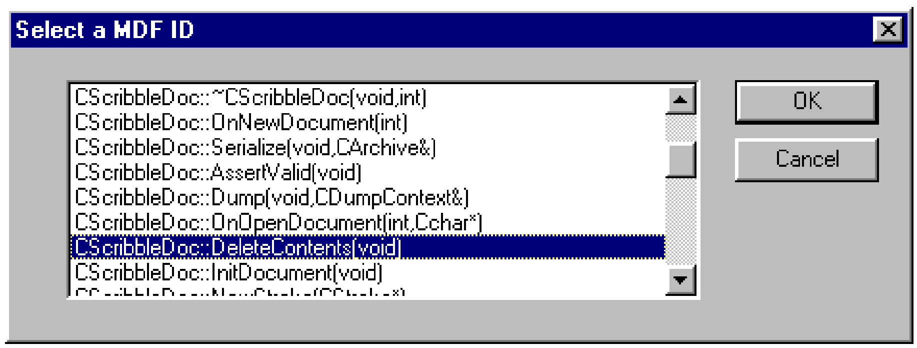


FIGURE 22 Select MDF ID Box

A calltree depicting the selected function appears.

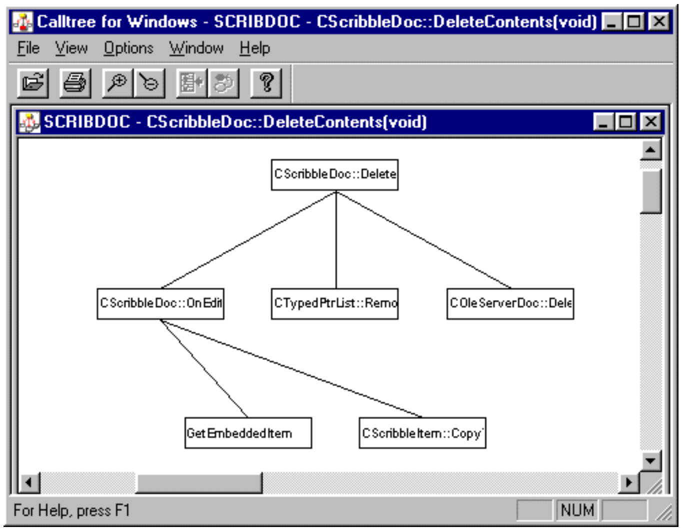


FIGURE 23 Displaying a Calltree

The calltree shows all of the callpairs associated with the function **CScribbleDoc::DeleteContents[void]**.

The next step shows how to look at digraphs of the possible program flows belonging to this function.

3.1.8 Step 7 - Viewing the Directed Graph Associated With a Calltree Node

To display a directed graph of any callpair shown in the calltree:

1. Select a node by clicking on it.

Notice that the **View Digraph** button on the toolbar now has a red arrow, indicating that it is available.

2. To display a directed graph of the selected function, click the **View DiGraph** button. You will see a directed graph of the **CScribble-Doc::DeleteContents[void]** function.

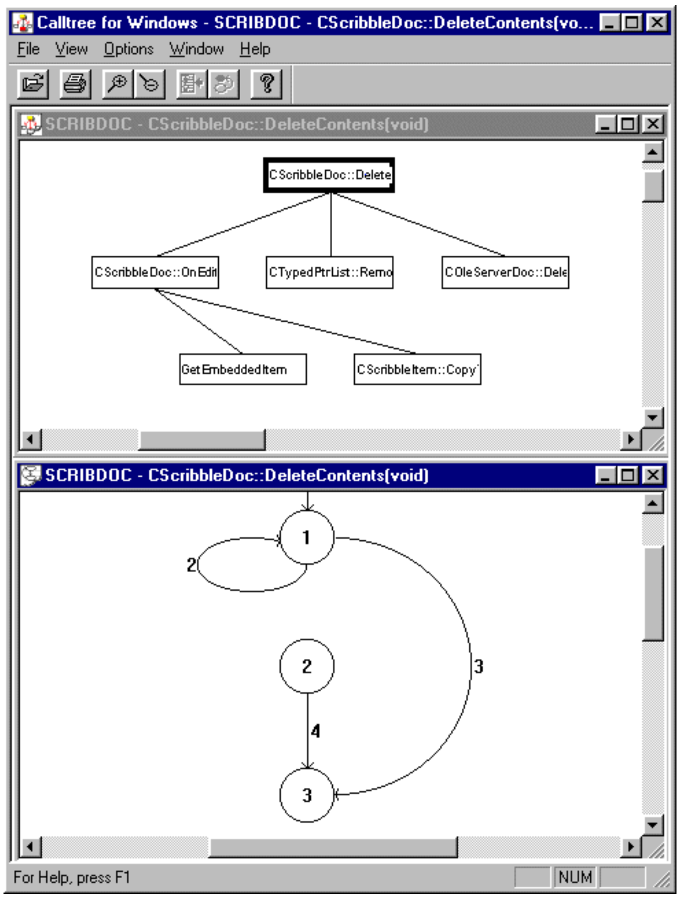


FIGURE 24 Calltree of **CScribbleDoc::DeleteContents[void]** and Digraph of Its Possible Program Flows

3.1.9 Step 8 - Viewing the Source Code Associated With a Calltree

You can view the source code associated with any node in a calltree by clicking on the corresponding edge.

Notice that the **Source Code** button on the Tool Bar has a red arrow.

1. To display the associated source code, click the Source Code button.

   The code is displayed in a separate window with the calling statement highlighted in red.

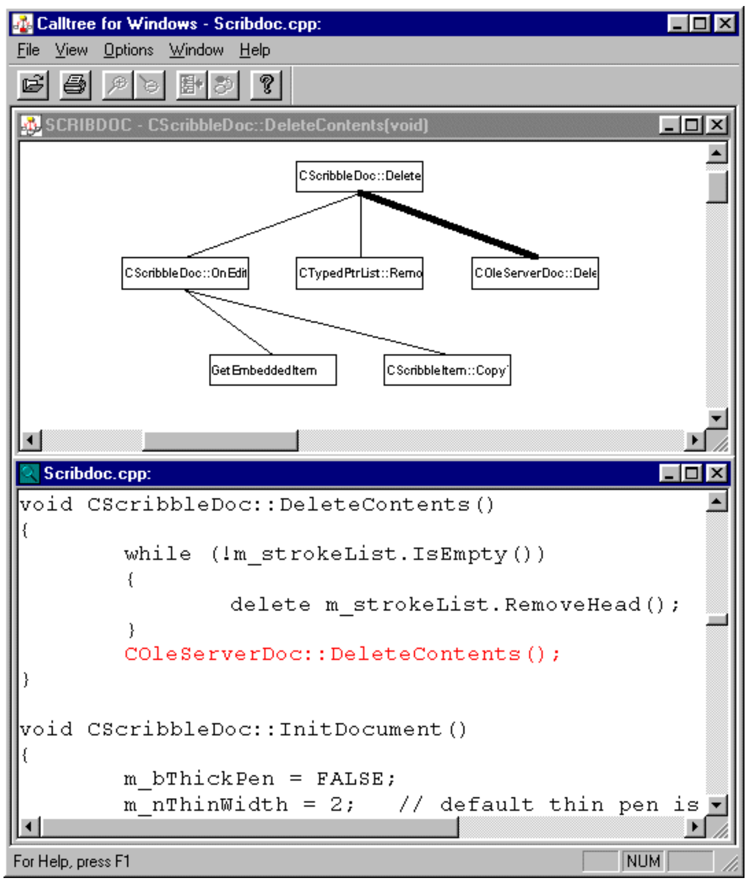


FIGURE 25 Source Code Window Displayed from Calltree

3.1.10 Step 9 - Closing TCAT C/C++for Windows

After looking at the source code, select one of the following options to complete the session.

To close **TCAT C/C++ for Windows:**

- Select **File|Exit** from the menu bar of each open program, or
- Double-click on the frame window **Close Box** of each program.

You have now seen all the main features of **TCAT C/C++ for Windows**.

3.2 Summary

If you have completed the proceeding steps successfully, you have seen and practised the basic skills you need to use TCAT C/C++ productively. You should have learned how to invoke TCAT C/C++, how to instrument, compile, link and run a program, and how to look at the coverage reports.

For best learning you may want to:

- Repeat STEPS 1 - 9 without the manuall and experiment by running the application several times and looking at the amount of coverage your test input receives.
- Repeat STEPS 1 - 9 with you application
- Review the chapters on system operation where you had difficulties. The table of contents can help you locate the topic you want.

CHAPTER 4

C/C++ Instrumentor Engine

This chapter discusses the **TCAT C/C++ for Windows** integrated “C” and “C++” instrumentor. This chapter applies to all editions of **TCAT C/C++ for Windows**.

4.1 Instrumentor Description

WinIC9 instruments the source code of the application under test by inserting function calls at each logical branch and call pair. The instrumentation does not affect the functionality of the program. When compiled, linked, and executed, the instrumented program will behave normally, but writes coverage data to a trace file.

There is some performance overhead related to the data collection process, but the overhead varies with the choice of the runtime used. The trace files are processed by several kinds of report generators.

There is a single version of the instrumentor engine for “C” and “C++” programs.

4.1.1 Files Generated

In operation, the **IC9** instrumentor parses candidate source code looking for logical branches and / or call pairs and generates auxiliary files that are used by other parts of the system. **TCAT C/C++ for Windows** uses and produces the following files:

Instrumenting Scribble.cpp produces the following files in the Example directory:

- *SCRIBBLE.i* — the instrumented version of the source file

  This file is updated during the instrumentation process.

- *SCRIBBLE.dg* — a Directed Graph Listing file

  Each instrumented file should have its own *.dg* file.

- *SCRIBBLE.cg* — a Calltree Graph Listing file

  Each instrumented file should have its own *.cg* file.

- *SCRIBBLE.mdf* — a Module Definition file

  This file contains information about segments and callpairs in all the processed files.

- *SCRIBBLE.obj* — the instrumented object file

  If you are working in a 32-bit environment, this file must be copied into the Debug directory.

  There is also a “C” version of this same information set up as a “C” structure format so that it can be used in cross-testing and embedded applications.

- *Trace.trc* — produced when the instrumented application is executed

  This file contains coverage information for the current test.

4.2 WinIC9 Main Window

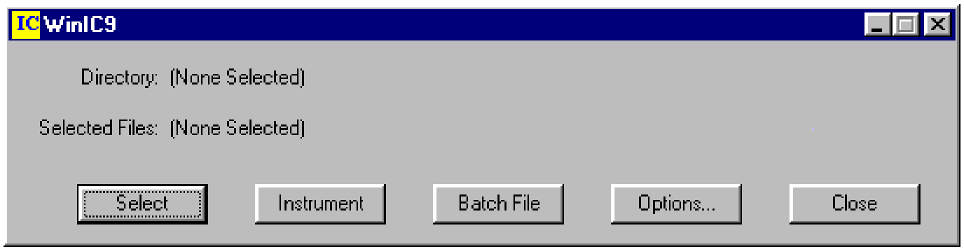


FIGURE 26 WinIC9

WinIC9 drives the instrumentor, **IC9,** according to selections made by the user.

Select	Click a file to select it for instrumentation, control-click to select several files, or shift-click to select a series of files.
Instrument	Instruments the selected file(s). During instrumentation, a command-line box gives informational and warning messages.
Batch File	Click this button to run **WinIC9** on the file appearing in the file selection area.
Options	Selects among code languages and modes of instrumentation.
Close	Exits **WinIC9**.

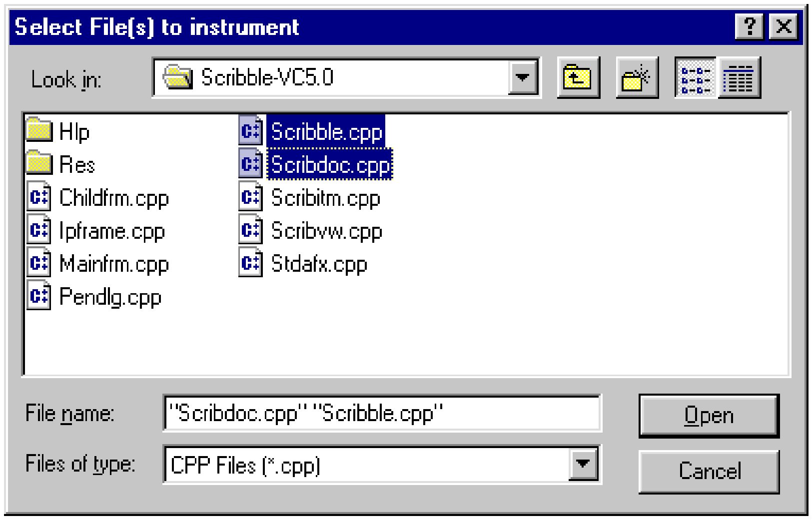


FIGURE 27 Select File(s) to Instrument

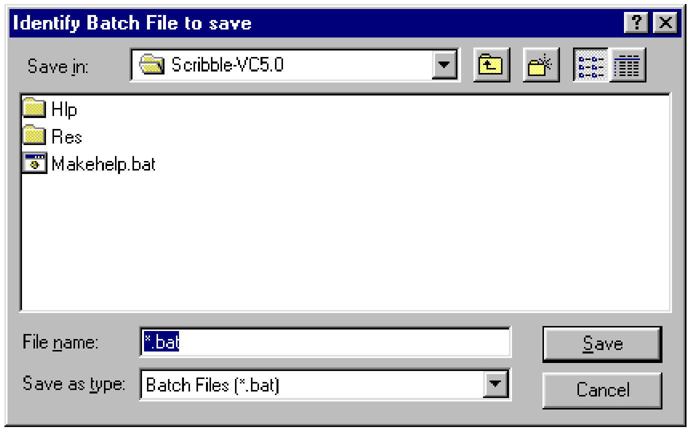


FIGURE 28 Identify Batch File

This option defers instrumentation. Thus, the batch file can become part of other time-consuming processes normally done overnight, such as fetching code or compiling big projects. When a **.bat* file is executed, it checks the interactive option and switches it off.

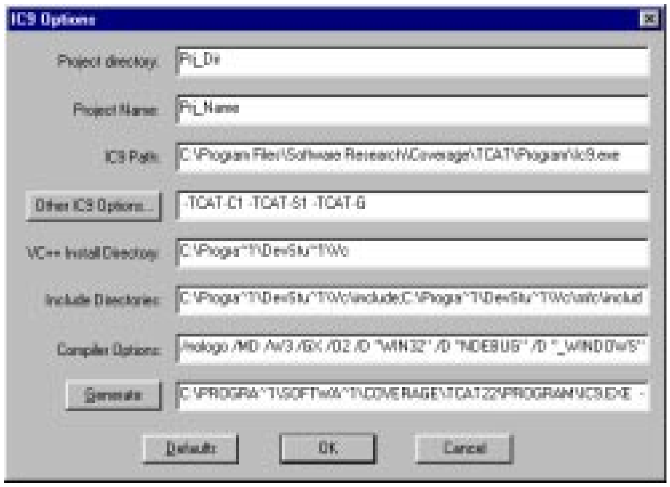


FIGURE 29 IC9 Options

Figure 23 shows the default options for **IC9**.

On 32 bits, any alterations generated here are written to the Registry key *HKEY_CURRENT_USER \ Software \ Software Research \ Coverage \ TCAT \ program \ WinIC9*, from which WinIC9 reads them. The Defaults button retrieves the contents of Registry key *HKEY_LOCAL_MACHINE \ SOFTWARE \ Software Research \ Coverage \ TCAT \ Program \ WinIC9* to this box.

For the options offered under Code Recognition, the C languages are optional; C++ is the default, and is recommended for use even with C files. Some C files contain constructs that might compile in C but not in C++; but absent these constructs, the C++ default is superior to the C options.

For the Instrumentation options, the usual assumption is that more coverage is better. Note that S0 coverage requires S1 coverage and cannot be selected unless S1 coverage is also selected.

Selecting the Keep Instrumented File option means that the **.i* file created during instrumentation is retained. Should the instrumentation fail, this file can be debugged for information, or compiled without using **IC9** to create **.obj* files.

Selecting the Instrument Only option prevents **IC9** from compiling and producing an **.obj* file.

The Interactive option makes the instrumentation more visible. The interactivity means that the **IC9** command line window, which is present during instrumentation, waits for the user to exit from it before closing down to begin instrumentation of the next file or to return to **WinIC9**. This ensures that the user can read the messages and warnings in the window. This option is automatically switched off for batch processing.

4.3 Instrumenting the Application Under Test

4.3.1 Options and Parameters

The syntax for command line invocation of **IC9** is as follows:

```
IC9 <<option>> file.ext
[-TCAT-A]
[-TCAT-B]
[-TCAT-Cmd driver]
[-TCAT-C1]
[-TCAT-E]
[-TCAT-G]
[-TCAT-H]
[-TCAT-K]
[-TCAT-O file]
[-TCAT-PD name]
[-TCAT-PN name]
[-TCAT-S0]
[-TCAT-S1]
[-Ddefs[=val]
[-Ipath]
[-Uundefs[=val]
```

These commands instrument submitted “C” and “C++” language file(s).

The directory specified with the -TCAT-PD switch becomes the project directory for the instrumentation. Within this directory, the t*cat_db* directory is automatically created. The directory name specified with the -TCAT-PN switch is created under the *tcat_db* directory, and contains the trace file, the module definition file, and the *c_graph* and *d_graph* directories. These lowest directories contain the *.*cg* and *.*dg* files, respectively.

If you invoke **IC9** with the switches `-TCAT-PD c:\AAA` and `-TCAT-PN XXX` on the file *`example.c`*, the directory tree created during instrumentation is as follows:

```
c:\
└──→AAA
     └──→tcat_db
          └──→ XXX
               ├──→c_graph
               │    └──→example.cg
               ├──→d_graph
               │    └──→example.dg
               ├──→XXX.mdf
               └──→trace.trc
```

The following instrumentor switches may be used to vary the processing and reports generated by the instrumentor. The instrumentor switches are listed in alphabetical order.

Note that the commands are prefixed with -*TCAT*. This is done because all other switches are passed to the "C" or "C++" compiler. The prefix indicates that these switches are for TCAT processing.

file.ext	Instrumented File Specification(s); File(s) to be instrumented
	The extension can be **c** or **i** or **cpp** (for "C++").
	If there are multiple files, each one is processed in the order presented, and they are treated as if they have been concatenated together.
-TCAT-A	ANSI Recognition Switch
	If present, the instrumentor recognizes only the ANSI version of "C" or "C++".
-TCAT-B	Non-Interactive Instrumentation Switch
	Instrumentation does not require any input from tested even if more than one file is being instrumented.
-TCAT-Cmd driver	Compiler Driver Command Switch
	Default driver is **cc.** For Microsoft Visual C, use **cl.exe.**-*TCAT-C1* C1 Instrumentation Switch
	If this switch is present, then the instrumentor inserts a function call in each segment, or logical branch. This is the preset default.
-TCAT-E	Print Error Messages Switch
	This switch enables sending error messages to standard output. If not present, then error messages are suppressed.

-TCAT-G Instrumented File Disposition Switch

Normally the instrumentor does not keep the instrumented file, because it has already been used to produce the instrumented output. When this switch is present the instrumented files are retained.

-TCAT-Help Help Message Switch

This switch prints out the set of valid switches.

-TCAT-K K&R C Recognition Switch

If present, the instrumentor recognizes K&R "C".

-TCAT-i Instrumentation Only Switch

WinIC9 instruments the target application but does not generate an object file. *-TCAT-i* overrides the *-TCAT-cmd* switch.

-TCAT-O file Output File Specification

The output of the instrumentation process is directed to the named file (default is *file.i*).

-TCAT-PD name Project Directory Switch

This switch specifies the location of the "project" directory.

-TCAT-PN name Project Name Switch

This switch specifies the project name.

-TCAT-S0 S0 Instrumentation Switch

If this switch is present, then the instrumentor inserts a function call in each module. This tells you which functions are actually called during the invocation of the program, but it does not indicate the callee functions. To do this, you need to use the **-S1** switch.

-TCAT-S1 S1 Instrumentation Switch

If this switch is present, then the instrumentor inserts a function call in each call pair.

-Ddefs[=val] Establish Definition Switch

This switch establishes a definition that is passed on to the compiler.

-Ipath Include File Search Path Specification

This switch specifies the path on which to resolve the search for **#include** files.

-Uunefs[=val] De-Establish (Undefine) Definition Switch

This switch removes a definition that is passed on to the compiler.

4.3.2 Instrumentation Function Names

Instrumentation involves inserting function names into the source program. The function names for TCAT-instrumented programs are:

`SegHit();`	For entry segment, switch segments
`CprHit();`	For S1 coverage of call pairs
`ExpHit();`	For C1 coverage **if**'s, **while**'s and **for**'s
`Strace();`	Start trace operations (this is an optional call)
`Ftrace();`	Finish trace operations, flush buffer, and close trace file

NOTE: For console (non-GUI) applications in Windows 95 and Windows NT and applications targeted for DOS in Windows 3.1*x*, trace files cannot be created correctly if the **`main`** function contains a **`return`**. This is because **WinIC9** inserts **`Ftrace();`** following any instance of **`return`** in the **`main`** function of an instrumented program, which terminates the program before the trace file can be closed and the buffer flushed. If this happens, substituting **`exit`** for **`return`** in the **`main`** function averts the problem.

4.3.3 Instrumentor Inline Directives

It is possible to control instrumentation from within the processed “C” or “C++” file, using the following instrumentor directives to turn off/on all instrumentation (but keep the segments and call pairs numbered correctly):

```
/* TCAT OFF */
/* TCAT ON */
```

4.4 Database File Formats

For information on the format of **WinIC9** output files, see Appendix A, "C/C++ Instrumentor Engine Database Files."

CHAPTER 5

Cover

This chapter discusses **Cover**, the **TCAT C/C++ for Windows** complete TCAT C/C++ analyzer for branch (C1) and callpair (S1) metrics. This chapter applies to all editions of the product.

5.1 Cover

Cover analyzes the trace files created when an instrumented program is executed, and generates reports based on the trace file data.These coverage reports can be tailored to show a variety of data, including:

- segments hit
- segments not-hit
- past-test and cumulative coverage percentages

Cover makes the following assumptions:

- A [possibly empty] archive file and a current [possibly empty] trace file exist.
- There is a file containing the names of the files in the project.
- The actual update of trace + archive --> archive is optional at end of a session.

The package maintains its usual rules for precedence of archive over trace, and displays warning messages when it finds size differences between archive and trace file.

5.2 Trace File and Archive File Formats

For information on the format of trace files and archive files, see Appendix A, “C/C++ Instrumentor Engine Database Files.”

5.3 Cover Main Window

Once you have built an instrumented version of your application and exercised it, follow these steps to display a coverage report:

1. Click on **Cover** icon (C1) from the **MS-VC Studio** toolbar or from **Star -->Programs**, then select **TCAT C and C++** Program Group.
2. From the **File** menu, select **Open**.
3. In the **Open** dialogue box, click on the filename ***Trace.trc*** in the *tcat_db* directory. The dialog box then asks for an archive file; ignore this request by clicking the **Cancel** button.

A coverage report on the application appears.

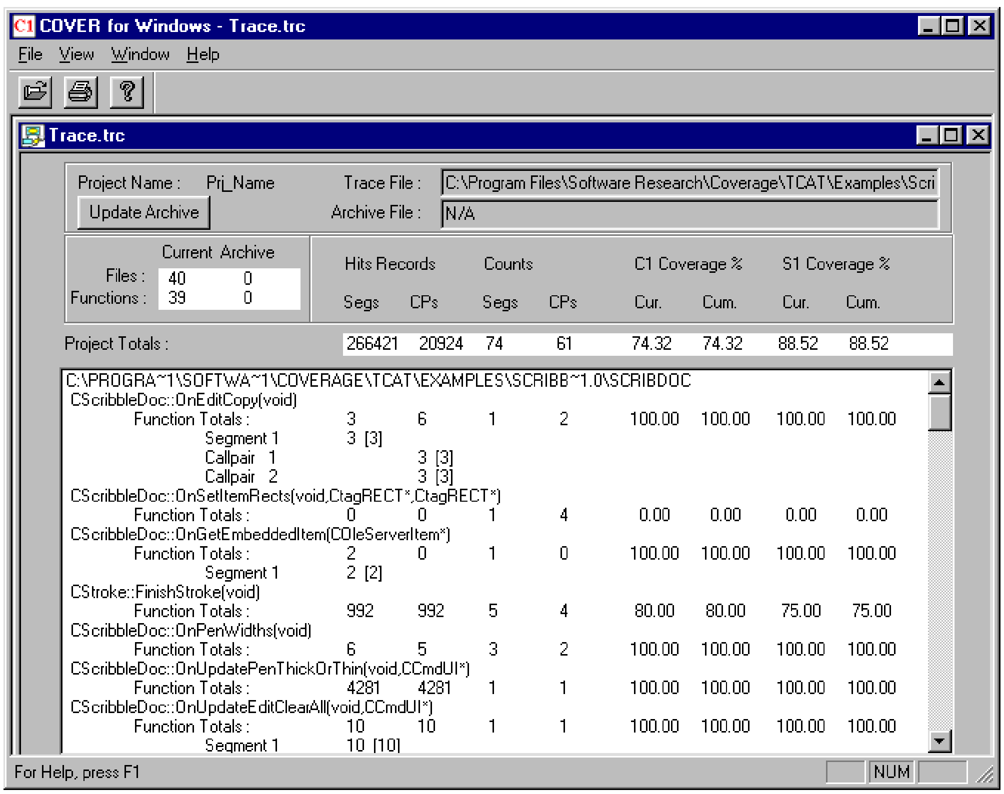


FIGURE 30 Cover Main Window

5.3.1 Tool Bar

The options available from the Tool Bar are the frequently used **Cover** features.

FIGURE 31 Tool Bar

Open	This option brings up the **Open** dialog box.
Print Button	This button brings up the **Print** dialog box.
Help	This button brings up a brief description of **Cover**.

5.3.2 File Menu

This menu displays the file management and printing options that are available in **Cover**.

Open This option brings up the **Open** dialog box.

Print This option brings up a the **Print** dialog box.

Print Preview This option displays an image of what prints when you select the **Print** option.

Print Setup This option displays a standard Windows printer set-up dialog box.

Exit To end your **Cover** session, select the **Exit** option.

5.3.3 View Menu

This menu provides two options for configuring the **Cover** display.

Toolbar — This toggle allows you to hide the Tool Bar in order to give your report more vertical display space or to re-display it.

Status Bar — This toggle allows you to hide or re-display the status bar at the bottom of the **Cover** window.

5.3.4 Window Menu

This menu allows you to manipulate the **Cover** windows using the **Cascade**, **Tile** and **Arrange Icons** options, and the **Window** list box.

5.3.5 Help Menu

The first help option currently offers a brief description of **Cover**. The second option, **About**, displays the program's version number and copy-right information.

5.3.6 Status Bar

This section of the window (appearing at the bottom left) displays messages regarding the functionality and operation of the **Cover** options.

5.4 File Menu

This menu is typical of Windows interfaces and provides access to file-manipulation options.

5.4.1 Open

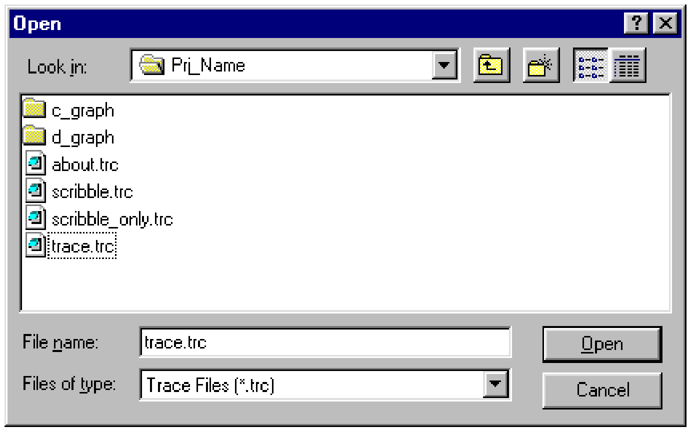


FIGURE 32 Cover Open Dialog Box

This option brings up a file selection dialog box. Typical of Windows interfaces, this dialog allows you to browse the directory tree and select files to open. Since all trace files are usually saved as *trace.trc*, each project has only one trace file.

File Name This box lists the files in the current directory that match the filter.

Directory This box lists the available directories.

When you have found the desired file, click **OK**, and the coverage report is displayed. **Cancel** closes the dialog box without opening a report.

5.4.2 Print

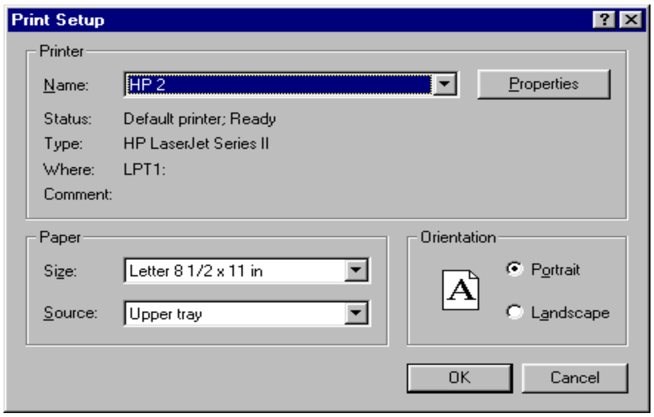


FIGURE 33 Print Dialog Window in Cover

The image you see is printed to a standard print device. Your printer may have different options. This window allows you to configure it for your environment. The following options are available in the **Print** dialog box:

Printer You must name the printer to which the printing of the document is to be sent.
When a print job has been sent, a message window saying **Print action completed** pops up. Click **OK** to close this window.

Print Range This option allows you to print the entire document or a subset thereof.

Print Quality This pull-down menu allows you to select the quality of the print job.

Copies This option allows you to specify the number of copies to print. The **Collate Copies** check-box defaults to **Yes**.

There are four buttons available on this dialog box.

OK This button sends your print job to the specified printer.

Cancel This button closes the dialog box without printing your document.

Printer Setup The button opens the Printer Setup dialog box, where you can select a printer and change printing options.

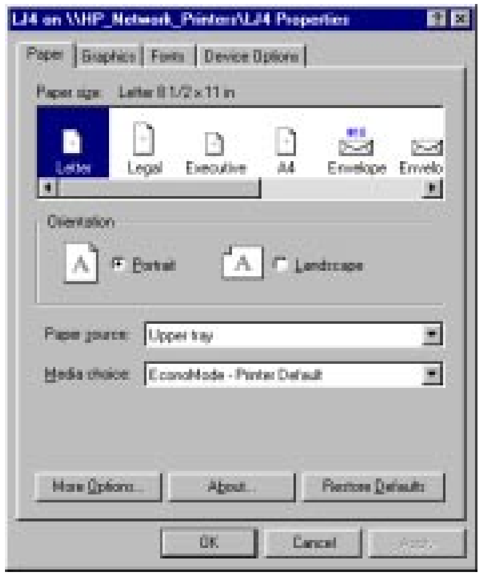


FIGURE 34 Print Setup Dialog

5.5 Window Menu

This menu provides four options to manipulate the **Cover** windows. By default the active window entirely overlaps all others.

5.5.1 Cascade

This option arranges your windows in a cascade, with the active window top-most and highlighted.

5.5.2 Tile

This option arranges the windows so that a portion of each window is displayed. The active window is highlighted.

5.5.3 Arrange Icons

When you have minimized windows, this option arranges them neatly at the bottom of the **Cover** window.

5.5.4 Window List Box

This area of the pull down-menu lists all the windows open in **Cover**. The active window is indicated by a check mark. To activate a new window, especially if the windows are fully overlapping, select it from this list.

5.6 Create/Update an Archive File

If no archive file is loaded, this option creates one by copying the current **.trc* file as an **arh* file. Updating combines the information from the current **.trc* file with that of the selected **.arh* file.

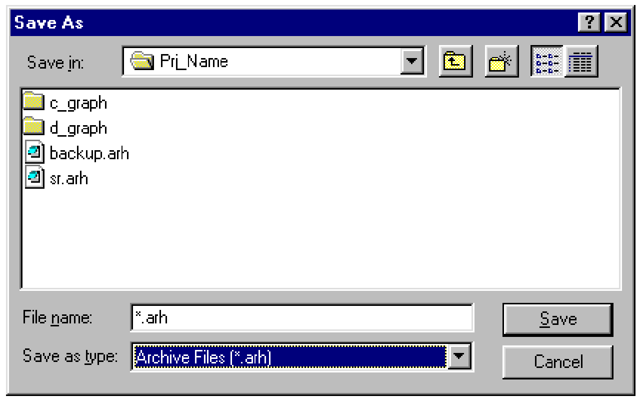


FIGURE 35 Save Archive File

5.7 Analysis of Coverage Reports

In the following analysis, a coverage report shows that a certain function, **CScribbleDoc::DeleteContents[void]**, has been tested 75.00%.

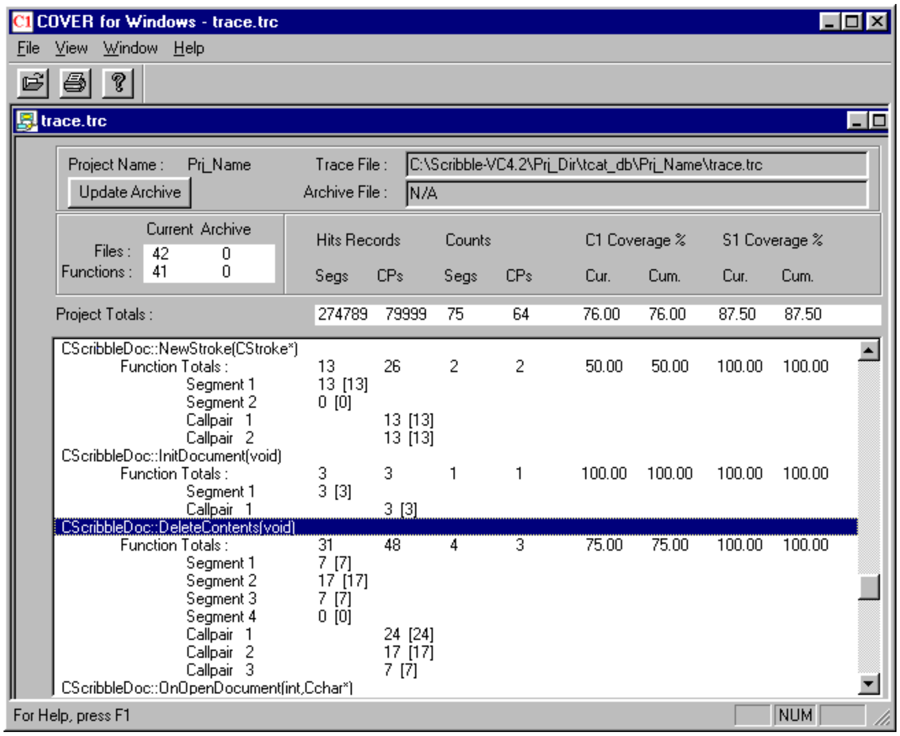


FIGURE 36 Coverage Report Showing C1 Coverage of 75.00% on the Function **CScribbleDoc::DeleteContents[void]**

The function consists of four segments and three callpairs. This coverage report shows that segments 1 and 3 were hit 7 times each, segment 2 was hit 17 times, and segment 4 not once. The callpairs 1 was exerised 24 times, callpair 2 was exercised 17 times, and callpair 3 was exercised 7 times.

The following few pages show graphical views of these numerical results.

In Figure 37, **TCAT C/C++ for Windows** graphs **CScribbleDoc::DeleteContents[void]** and its relations. The calltree shows the callpairs in **CScribbleDoc::DeleteContents[void]**, and the digraph shows possible program flows through **CScribbleDoc::DeleteContents[void]** divided into segments.

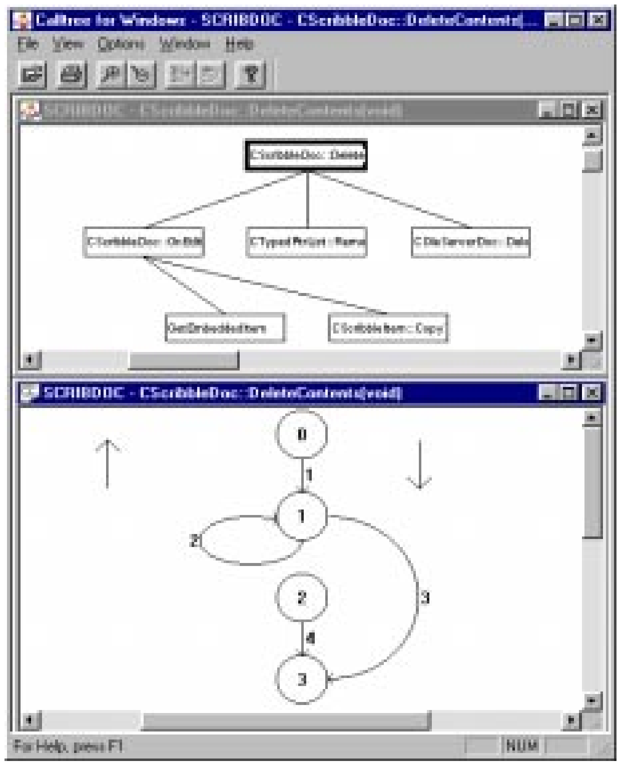


FIGURE 37 Calltree and Digraph of **CScribbleDoc::DeleteContents[void]**

Note that the calltree shows three callpairs: these callpairs are shown in the coverage report in Figure 36, which have been exercised 24, 17, 7 times respectively. The coverage report shows that the percentage of S1 coverage (coverage of call pairs) was 100% for this function.

Note that the digraph shows three segments. The coverage report in Figure 36 shows that the test of **Scribble** hit three of four segments. The coverage report shows that the percentage of C1 coverage (branch coverage) was 75.00%.

To look at source code associated with callpairs, highlight the graphic lines connecting the functions shown in the calltree.

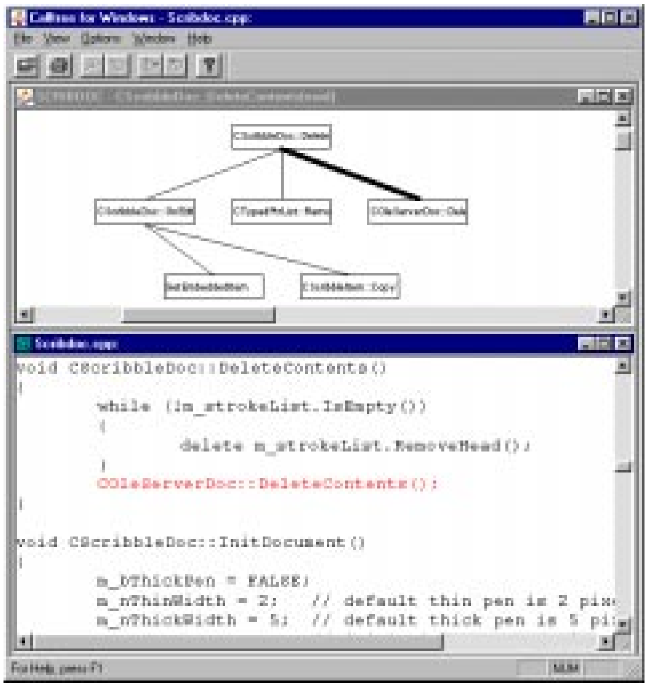


FIGURE 38 Calltree and Source Code Associated with One Callpair

To look more closely at the segments, highlight one of the graphic lines in the digraph by clicking on it close to the number. Then use the Source Code button to display the associated source code.

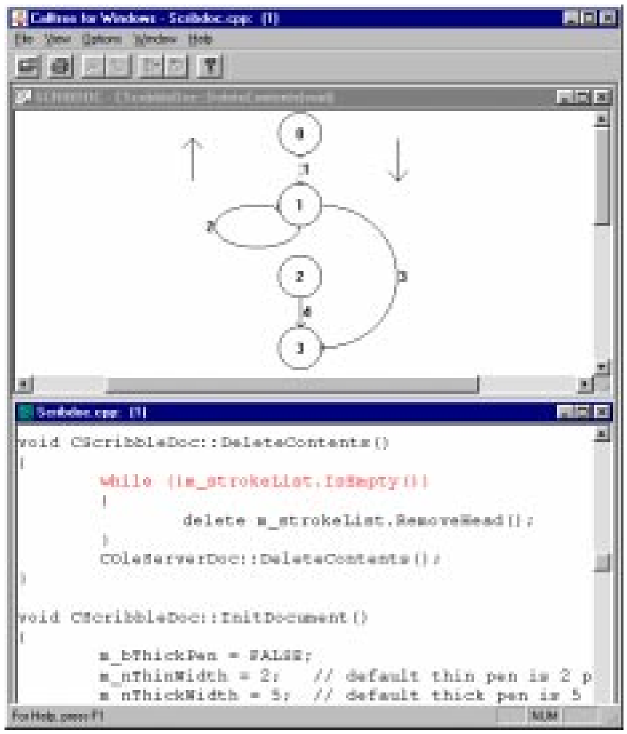


FIGURE 39 Digraph and Source Code Associated with One of Its Segments

CHAPTER 6

DiGraph

This chapter provides details on viewing and using directed graphs in **TCAT C/C++ for Windows**.

6.1 Purpose and Overview

Directed graphs (digraphs) graphically display a program's structure and flow to help developers isolate flaws and bottlenecks.

TCAT C/C++ for Windows draws digraphs based on archive files that are created during instrumentation. Digraphs are composed of **edges** and **nodes**. Edges are derived from segments (also known as logical branches) representing sets of consecutive program statements or a program's "actions" (see Figure 40). Nodes are the places or "states" where the actions occur.

6.2 Directed Graph File Format

For information regarding the format of a directed graph chart file, see Appendix A, "C / C++ Instrumentor Engine Database Files."

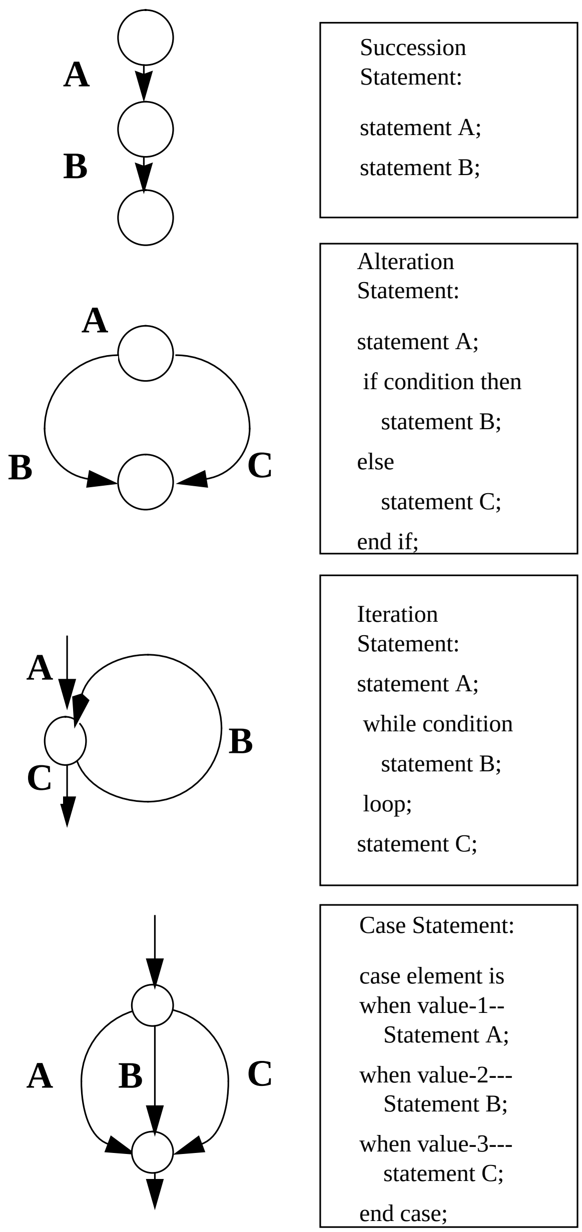


FIGURE 40 Program Edges as Represented in a Digraph

6.3 DiGraph Main Window

In order to explore all the options available, open a directed graph of the example program. In order to do this, you must first instrument the example application, which is discussed in Sections 4.1, 4.2, and 4.3, "Using IC9."

When you have an instrumented executable:

1. Click on the **DiGraph** icon from the **MS-VC Studio** toolbar or from **Star -->Programs**, then select **TCAT C and C++** Program Group.
2. Using the **File** pull down menu and select **Open**.

   You are prompted for the name of the directed graph to view.
3. Find the *SCRIBBLE.dg* file under the *tcat_db \ name \ d_graph* directory.

   You are prompted for the name of the database file.
4. Find the *SCRIBBLE.mdf* file under the *tcat_db \ name* directory.

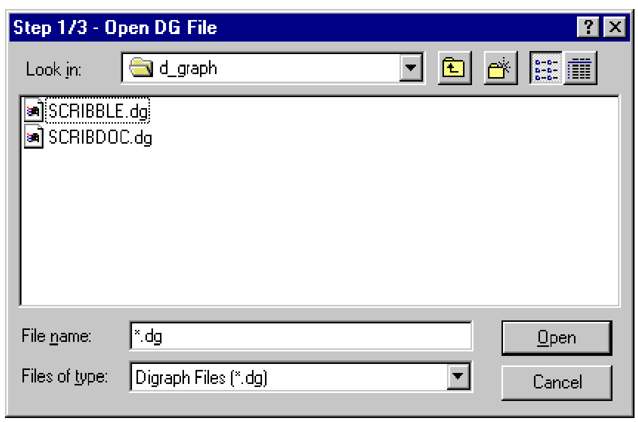


FIGURE 41 WinDiGraph Open Dialog Box

5. A window pops up listing the available functions (Figure 42). For this example, select **CScribbleDoc::DeleteContents[void]**.

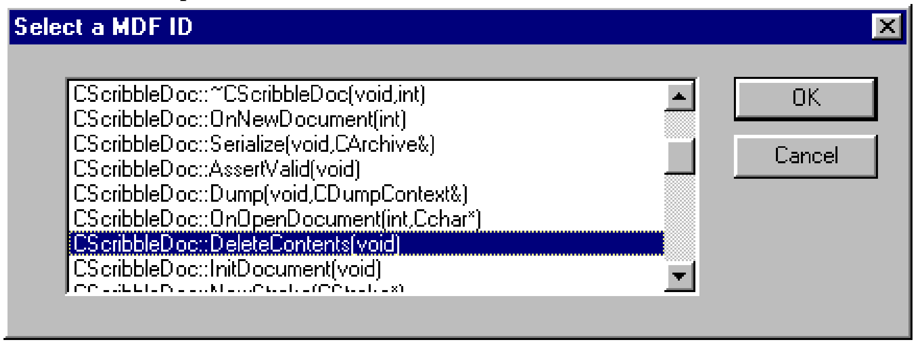


FIGURE 42 Select MDF ID Box

A directed graph depicting possible program flows of the function **CScribbleDoc::DeleteContents[void]** appears.

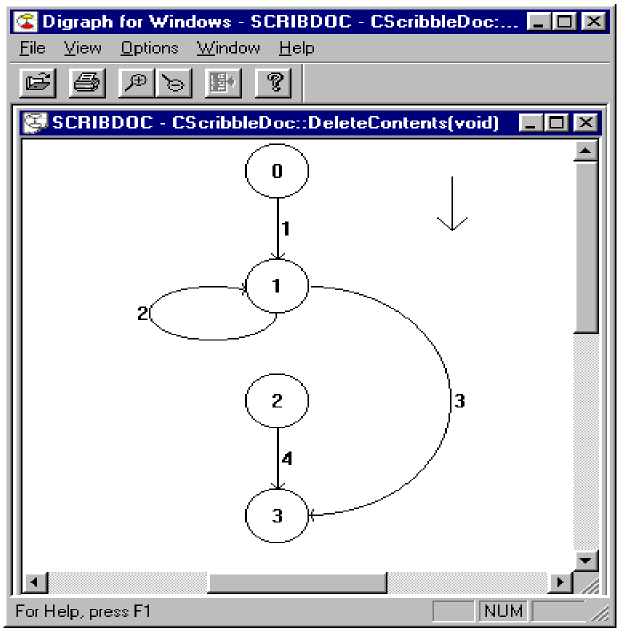


FIGURE 43 Directed Graph of Scribble

The digraph shows the set of conditions and paths that make up a function. The next step shows how to look at the code that the digraph displays as numbered segments.

6.3.1 Tool Bar

The options available from this Tool Bar are the frequently used DiGraph features. When available, they appear highlighted.

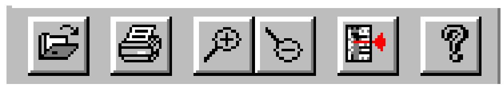


FIGURE 44 Tool Bar

Open	This button brings up the **Open** dialog box.
Print	This button brings up the **Print** dialog box.
ZoomIn	This button Zooms in magnification factors of the current open window.
ZoomOut	This button Zooms out magnification factors of the current open window.
Source	This button brings up a window which contains the source code for the currently selected edge.
Help	This button brings up a brief description of DiGraph.

6.3.2 File Menu

This menu displays the file management and printing options that are available in **DiGraph**.

Open This option brings up the **Open** dialog box.

Print This option brings up a the **Print** dialog box.

Print Preview This option displays an image of what will print when you select the **Print** option.

Print Setup This option displays a standard Windows printer set-up dialog box.

Exit To end your **DiGraph** session, select the **Exit** option.

6.3.3 Zoom Menu

This menu contains two options for scaling the digraph's display. For information on setting the zoom scale, see Section 6.6.1, "The Digraph Options Dialog Box."

In This option allows you to enlarge a portion of the digraph so that you can see it in more detail. There is a limit to how far you can zoom in, determined by your computer's display resolution.

Out This option allows you to see a wider portion of the digraph at a reduced magnification. Again, limits apply to how far you can zoom out.

6.3.4 View Menu

This menu provides three options for configuring the digraph's display.

Source This option allows you to display the source code for the selected function in the current directed graph.

Tool Bar This toggle allows you to hide the Tool Bar in order to give your digraph more vertical display space or to re-display it.

Status Bar This toggle allows you to hide or re-display the status bar at the bottom of the **DiGraph** window.

6.3.5 Options Menu

This menu provides access to two dialog boxes where you can set global display options for **DiGraph**.

Digraph Options This option displays a dialog box allowing you to choose the characteristics of the nodes and edges displayed in the digraph, as well as the increments for the **Zoom In** and **Zoom Out** options.

6.3.6 Window Menu

This menu allows you to manipulate the **DiGraph** windows using the **Cascade, Tile,** and **Arrange Icons** options, and the **Window** list box.

6.3.7 Help Menu

The first help option currently offers a brief description of **DiGraph**. The second option, **About**, displays the program's version number and copyright information.

6.3.8 Status Bar

This section of the window (appearing at the bottom left) displays messages regarding the functionality and operation of the **DiGraph** options.

6.4 File Menu

This menu is typical of Windows interfaces, and provides access to file-manipulation options.

6.4.1 Open

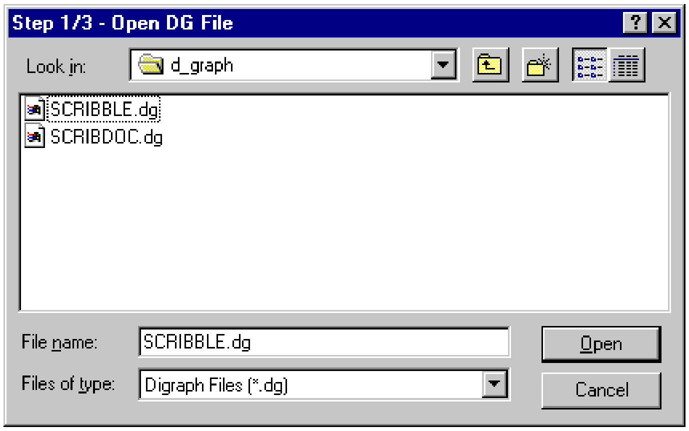


FIGURE 45 DiGraph Open Dialog Box

This option brings up a file selection dialog box. Typical of Windows interfaces, this dialog box allows you to browse the directory tree, and select files to open.

File Name This box lists the files in the current directory that match the filter.

Directory This box lists the available directories.

When you have found the desired file, click **OK**, and the directed graph is displayed. **Cancel** closes the dialog box without opening a graph.

6.4.2 Print

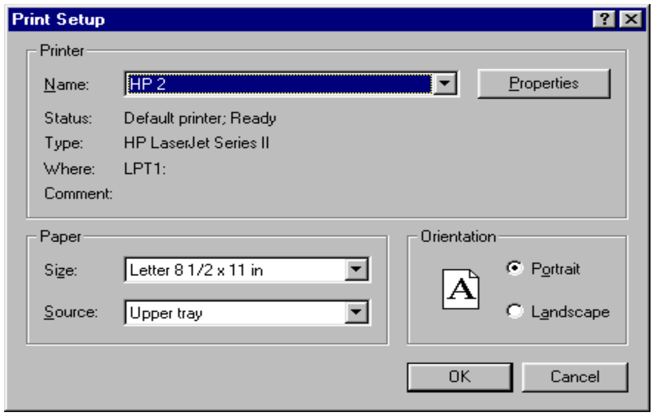


FIGURE 46 Print Dialog Box in DiGraph

The image you see is printed to a standard print device. Your printer may have different options. The following configuration options are available in the Print dialog box:

Printer — You must name the printer to which the printing of the document is sent.
When a print job has been sent, a message window saying **Print action completed** pops up. Click **OK** to close this window.

Print Range — This section allows you to print the entire document, or a subset thereof.

Print Quality — This pull down menu allows you to select the quality of the print job.

Copies — This option allows you to specify the number of copies to print. The **Collate Copies** check-box defaults to **Yes**.

There are four buttons available on this dialog box.

OK — This button sends your print job to the specified printer.

Cancel — This button closes the dialog box without printing your document.

Printer Setup This button opens the Printer Setup dialog box where you can select a printer and change printing options.

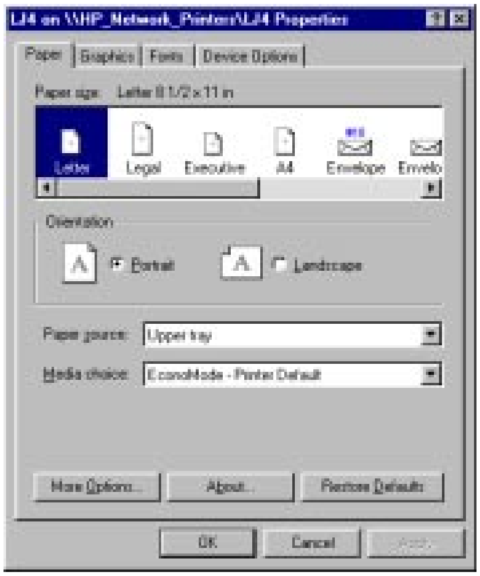


FIGURE 47 Print Setup Dialog Box

6.5 View Menu

The most critical option on this menu is the **View Source** option.

6.5.1 Viewing Associated Source Code

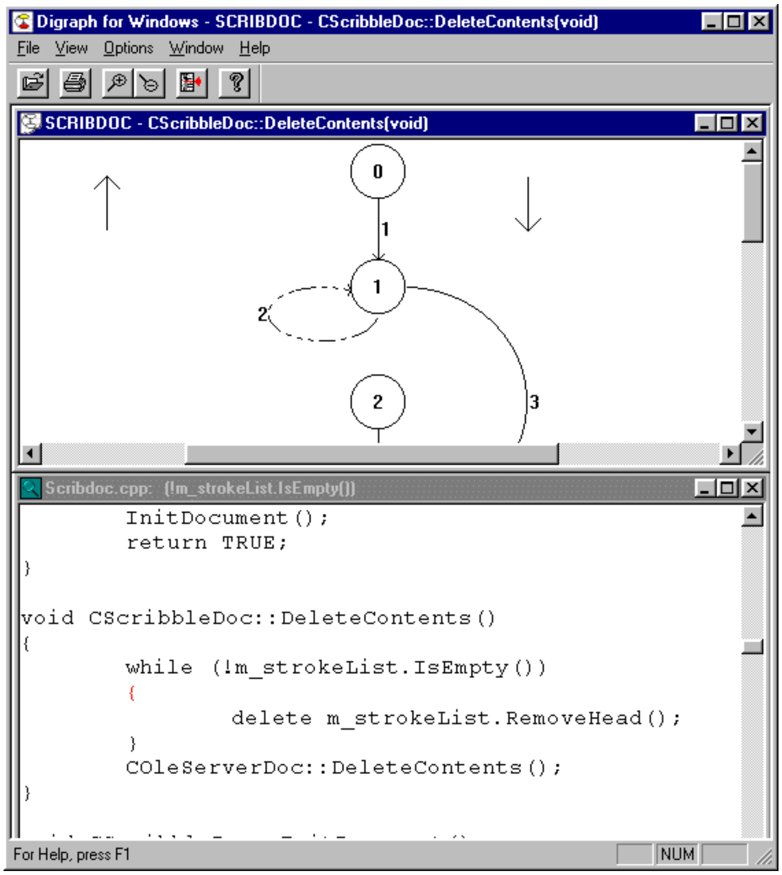


FIGURE 48 View Source Option

This option displays the source code for the program depicted in the digraph. If you click on an edge segment number in the digraph's main window, and the **View Source** option, the source code associated with that edge is displayed.

The arrow (triangle) symbols on the right-hand side (and bottom, when appropriate) of the window are scroll bars, which you can use to move vertically (or horizontally) in this window.

6.6 Options Menu

The options available from this menu allow you to configure certain aspects of the **DiGraph** display.

6.6.1 The Digraph Options Dialog Box

Characteristics

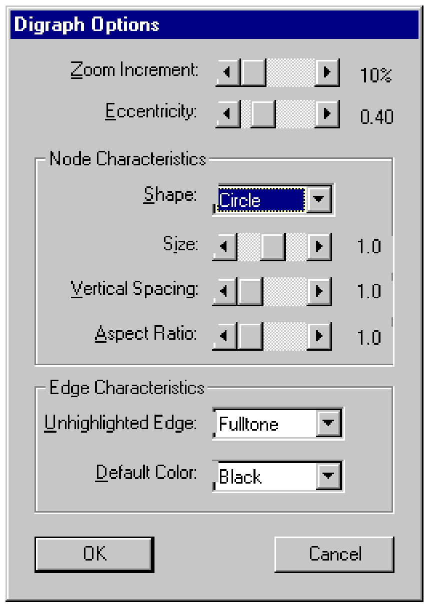


FIGURE 49 Digraph Options Dialog Box

This dialog box allows you to choose the magnification step used for the **Zoom In** and **Zoom Out** commands, the shape and size of the digraph's nodes, and the colors of the digraph's edges.

Zoom Increment This sets the magnification interval for the **Zoom In** and **Zoom Out** options. The default setting is .1 meaning a 10% reduction or enlargement in scale each time these buttons are used. To change the setting, move the slider left or right. Each 0.1 represents 10%, so if you slide the rule to .3, for example, the reduction and enlargement is 30% each time.

Eccentricity This determines the curvature of the generated display. The default value is .3; bigger values make the picture wider, and smaller values narrower.

Node

You can choose different sizes and shapes for the digraph's nodes. In this window, you can change the space between nodes and their height-to-width ratio.

You have four choices for shapes: **Circle**, **Box**, **Oval** or **Outlined** (the circle is drawn but not filled). The default setting is **Circle**.

- You can choose the size of the circle, box or oval. The default size is 1.0.
- You can change the amount of space between nodes. The default setting is 1.0.
- You can change the height-to-width ratio (for ovals or box shapes only). The default setting is 1.0.

Edge This area provides options to change the appearance of edges on your directed graph.

- There are three choices for Unhighlighted Edge: **Fulltone**, **Halftone** (dashes) or **Blank** (no visible lines). The default setting is **Fulltone**.
- Default Color is the basic color of the digraph's edges and nodes. The default setting is **blue**.

OK If you click on the **OK** button, all the current settings in the **Options** window are applied to the digraph.

Cancel If you click on the **Cancel** button, any changes you have made since opening the **Options** window are discarded.

Close If you click on the **Close** button, you exit the **Options** window.

6.7 Window Menu

This menu provides four options to manipulate the **DiGraph** windows. The default arrangement is that the active window entirely overlaps all others.

6.7.1 Cascade

This option arranges your windows in a cascade, with the active window top-most and highlighted.

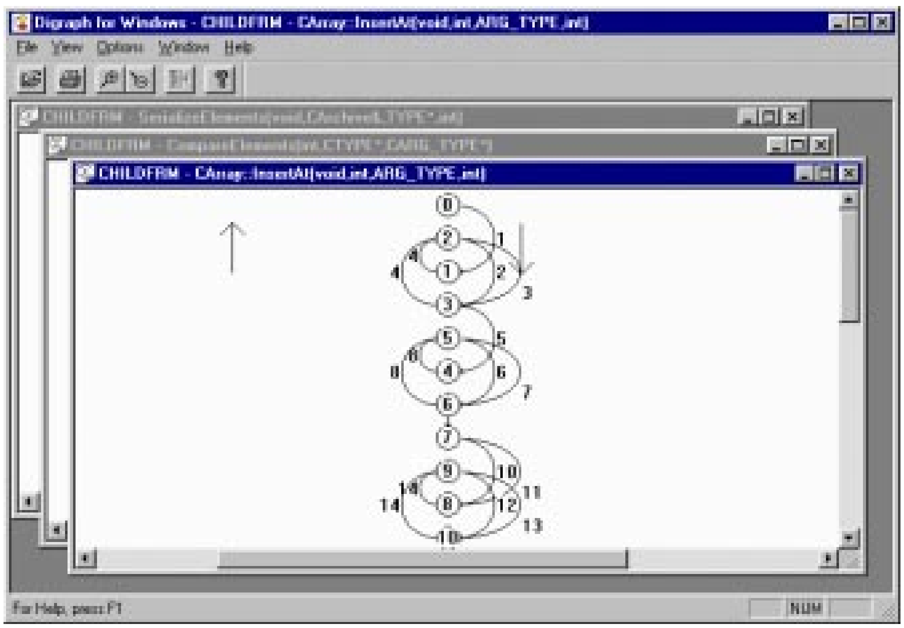


FIGURE 50 Cascading Windows in DiGraph

6.7.2 Tile

This option arranges the windows so that a portion of each window is displayed. The active window is highlighted.

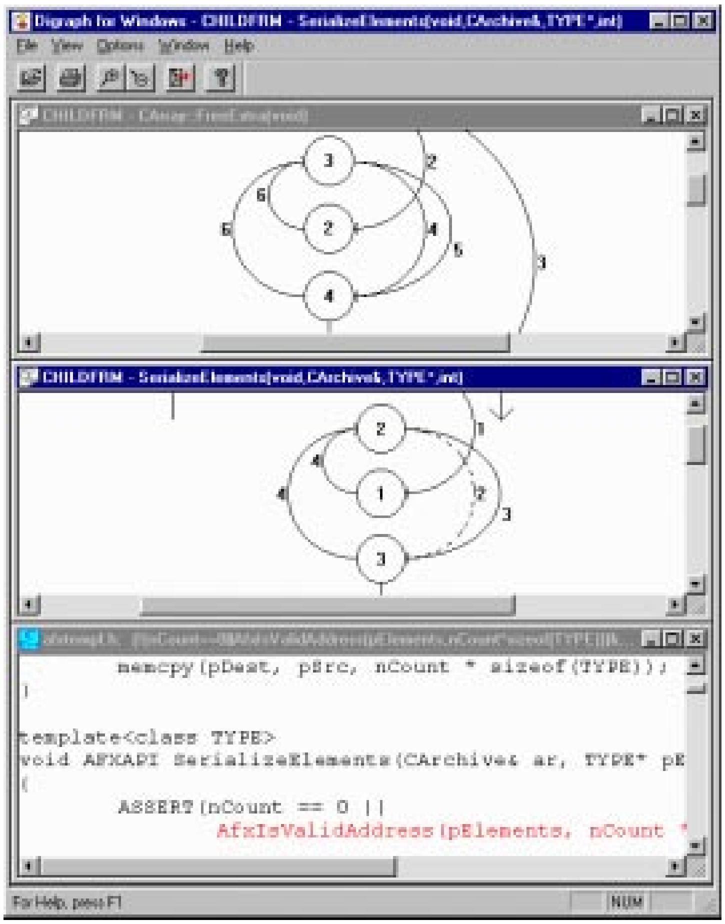


FIGURE 51 Tiled Windows in DiGraph

6.7.3 Arrange Icons

When you have minimized windows, this option arranges them neatly at the bottom of the **DiGraph** window.

6.7.4 Window List Box

This area of the pull down menu lists all the open windows available in **DiGraph**. The active window is indicated by a check mark. To activate a new window, especially if the windows are fully overlapping, select it from this list.

CHAPTER 7

CallTree

This chapter provides details about using calltrees in **TCAT C/C++ for Windows**.

7.1 Calltree Overview

A calltree displays a program's caller–callee dependency structure. **TCAT C/C++ for Windows** generates a calltree graph for each segment of your executable during instrumentation and stores it in a separate archive file. Once the instrumented application has been exercised, you can display a calltree window for a specified program segment by opening the target application's ***.cg** file.

7.2 Generating and Viewing Calltrees

You generate calltrees for your application by instrumenting your source-code files, as described in Sections 4.2 and 4.3 .

To Launch CallTree:

1. Click on the **CallTree** icon from the **MS-VC Studio** toolbar or from **Star -->Programs**, then select **TCAT C and C++** Program Group.

To View a calltree of the example program:

1. Pull down the **File** menu.
2. Select **Open**.

   You are prompted for the name of the calltree to view.
3. Find the *EXAMPLE.cg* file under the *tcat_db\name\c_graph* directory.

   You are prompted for the name of the database file.
4. Find the *TCAT.mdf* file under the *tcat_db\name* directory.
5. Select a function ID from the presented list.

A calltree depicting the selected function appears. This first node of the calltree is called the root, as it is never called from within the program. The second (and lower) tier of nodes are the called functions, as they are called by nodes above them. The final tier of a calltree consists of called functions which never call other functions.

7.3 Calltree File Format

For information on the format of calltree files, see Appendix A, "C/C++ Instrumentor Engine Database Files."

7.4 CallTree Window Overview

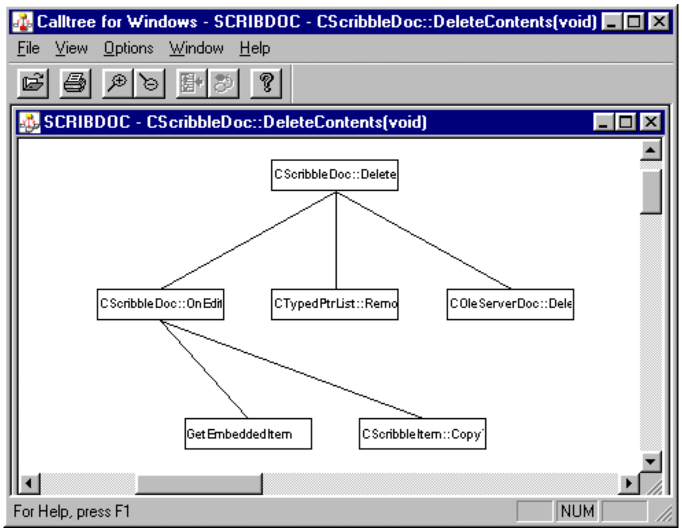


FIGURE 52 CallTree Main Window

This window allows you to view the calltree. This section briefly describes the menus available from **CallTree**. Several of the menus are discussed in more detail in later sections.

7.4.1 Tool Bar

The options available from this Tool Bar are the frequently-used CallTree features. When unavailable, they appear grayed out.

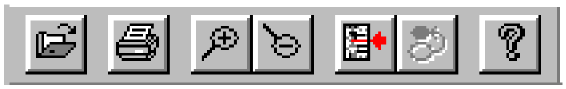


FIGURE 53 Tool Bar

Open	This button brings up the Open dialog box.
Print	This button brings up the Print dialog box.
ZoomIn	This button Zooms in magnification factors of the current open window.
ZoomOut	This button Zooms out magnification factors of the current open window.
Source	This button brings up a window which contains the source code for the currently selected edge.
Digraph	This button brings up a digraph of the associated function.
Help	This button brings up a brief description of CallTree.

7.4.2 File Menu

This menu displays the file management options available for CallTree.

Open — This option calls up the **Open** dialog box.

Close — This option closes the currently selected calltree.

Exit — If you wish to end your **CallTree** session, drag the mouse to **Exit**.

Print — This option brings up a the **Print** dialog box.

Print Preview — This option displays an image of what prints when you select the **Print** option.

Print Setup — This option displays a standard Windows printer set-up dialog box.

7.4.3 View Menu

This menu provides three options (**Select Function, Source** and **Directed Graph**) allowing alternate views of the program segment displayed in the calltree.

7.4.4 Window Menu

This menu allows you to manipulate any open **CallTree** windows using the **Cascade**, **Tile** and **Arrange Icons** options and the **Window** list box.

7.4.5 Options Menu

In this menu, a dialog box pops up where you can set the size, aspect ratio, and vertical spacing of the calltree, as well as the increments for the **Zoom In** and **Zoom Out** options.

7.4.6 Help Menu

This menu currently offers only one option, **About**, which displays the program's version number and copyright information.

7.4.7 Status Bar

This section of the window (appearing at the bottom left) displays messages regarding the functionality and operation of the CallTree.

7.5 File Menu

The File menu is typical of Windows applications.

7.5.1 Open

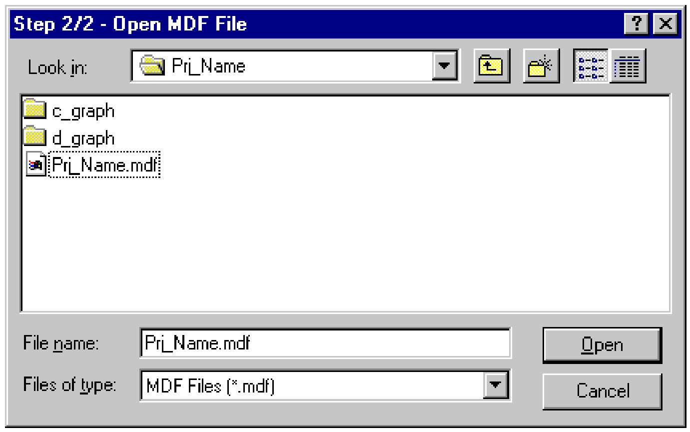


FIGURE 54 CallTree Open Dialog Box

This option brings up a file selection dialog box. It allows you to browse the directory tree and select files to open.

File Name This box lists the files in the current directory that match the filter.

Directory This box lists the available directories. When you have found the desired file, click **OK**, and the calltree is displayed.

Cancel closes the dialog box without opening a calltree.

7.5.2 Print Menu

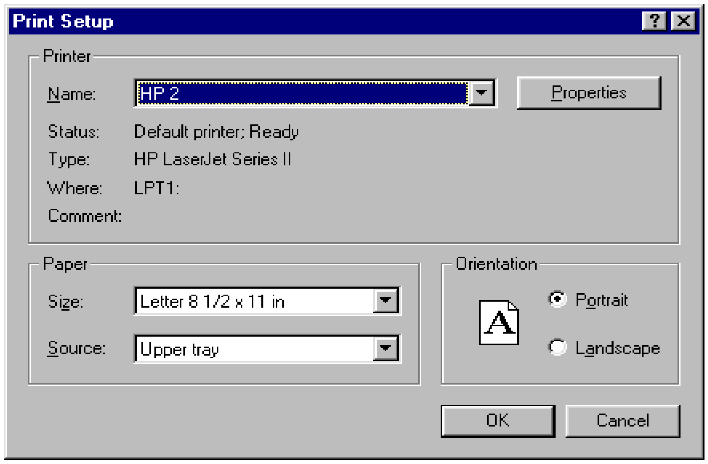


FIGURE 55 Print Dialog Box in CallTree

The image you see is printed to a standard print device. Your printer may have different options. The following configuration options are available in the Print dialog box:

Printer — You must name the printer to which the printing of the document is sent.
When a print job has been sent, a message window saying **Print action completed** pops up. Click **OK** to close this window.

Print Range — This section allows you to print the entire document or a subset thereof.

Print Quality — This pull-down menu allows you to select the quality of the print job.

Copies — This option allows you to specify the number of copies to print. The **Collate Copies** check-box defaults to **Yes**.

There are four buttons available on this dialog box.

OK — This button sends your print job to the specified printer.

Cancel — This button closes the dialog box without printing your document.

Printer Setup This button opens the Printer Setup dialog box, where you can select a printer and change printing options.

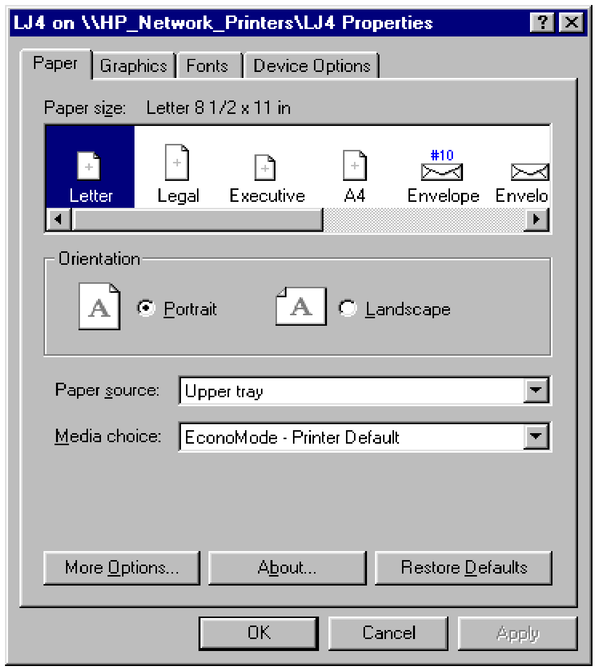


FIGURE 56 Print Setup Dialog Box

7.6 View Menu

From **CallTree**, you can view source code and directed graphs of your program using the options on this menu.

7.6.1 Viewing Associated Source Code

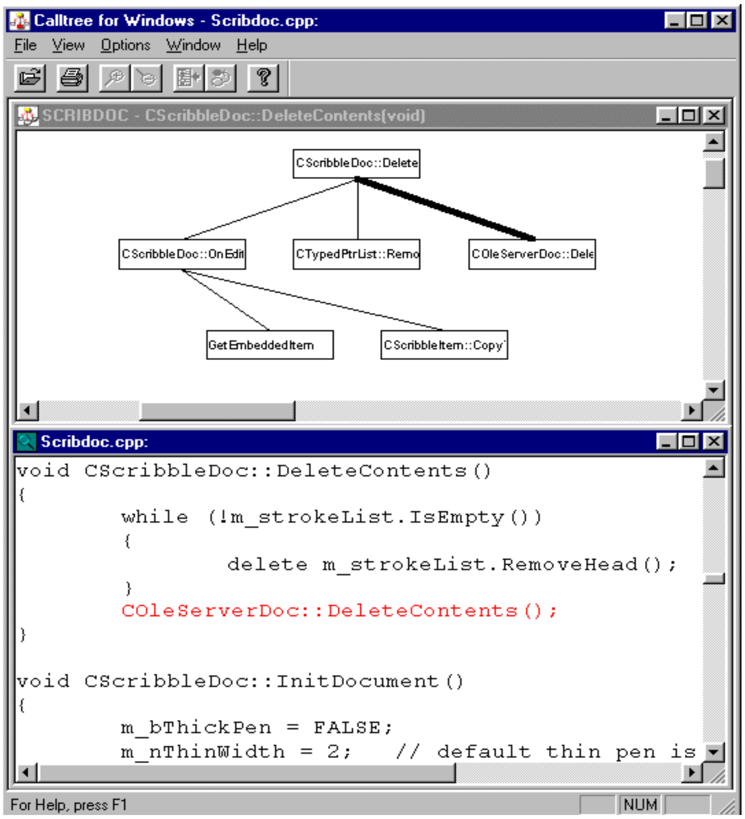


FIGURE 57 View Source Option

This option displays the source code for the program depicted in the call-tree. If you click on an edge segment in the calltree's main window, and select the **View Source** option, the source code associated with that edge is displayed. If no call pair was selected, the display is positioned at the first call pair in the module. You can also select the **Source** button on the Tool Bar.

The arrow (triangle) symbols on the right-hand side and bottom of the window are scroll bars, which you can use to move vertically or horizontally in this window.

7.6.2 Viewing a Directed Graph

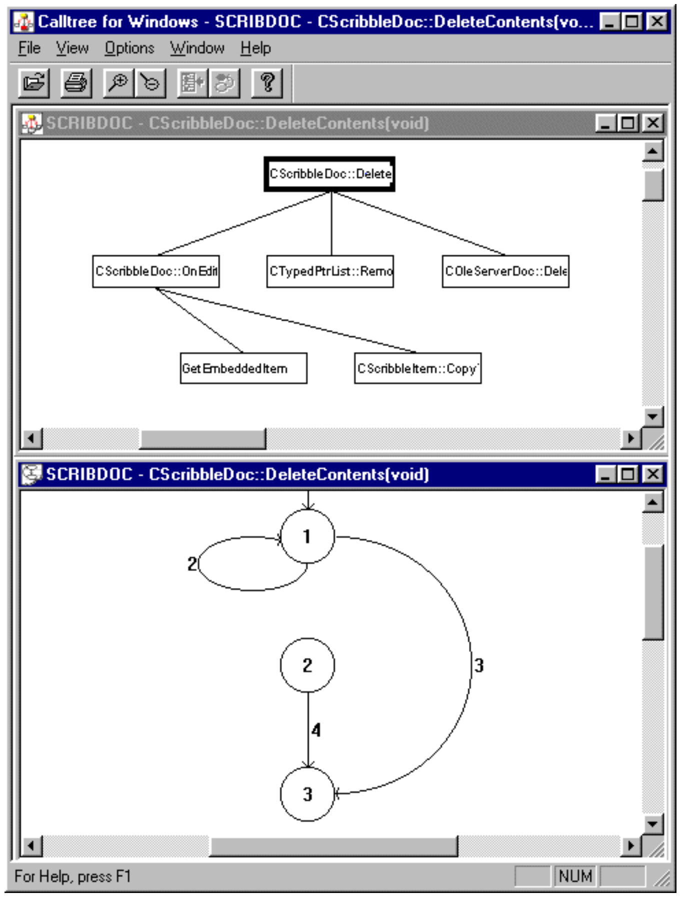


FIGURE 58 Directed Graph Option

This option allows you to view the detailed structure of a function in the current calltree. If you click on a node and select the **Directed Graph** option, a directed graph depicting that node appears. You can also select the **Directed Graph** button on the Tool Bar.

From this new window, you can view the source code in terms of edges and nodes rather than call pairs. To do so, click on an element of the directed graph and select **View Source** either from the **View** menu or from the Tool Bar.

7.7 Window Menu

This menu provides four options used to manipulate the **CallTree** windows.The default arrangement is that the active window entirely overlaps all others.

7.7.1 Cascade

This option arranges your windows in a cascade, with the active window top-most and highlighted.

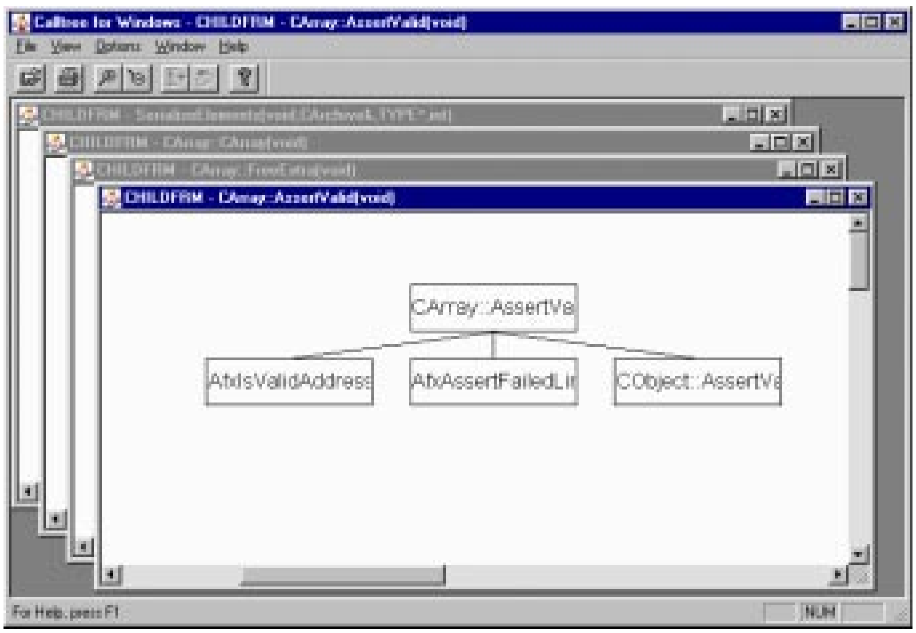


FIGURE 59 Cascading Windows in CallTree

7.7.2 Tile

This option arranges the windows so that a portion of each window is displayed. The active window is highlighted.

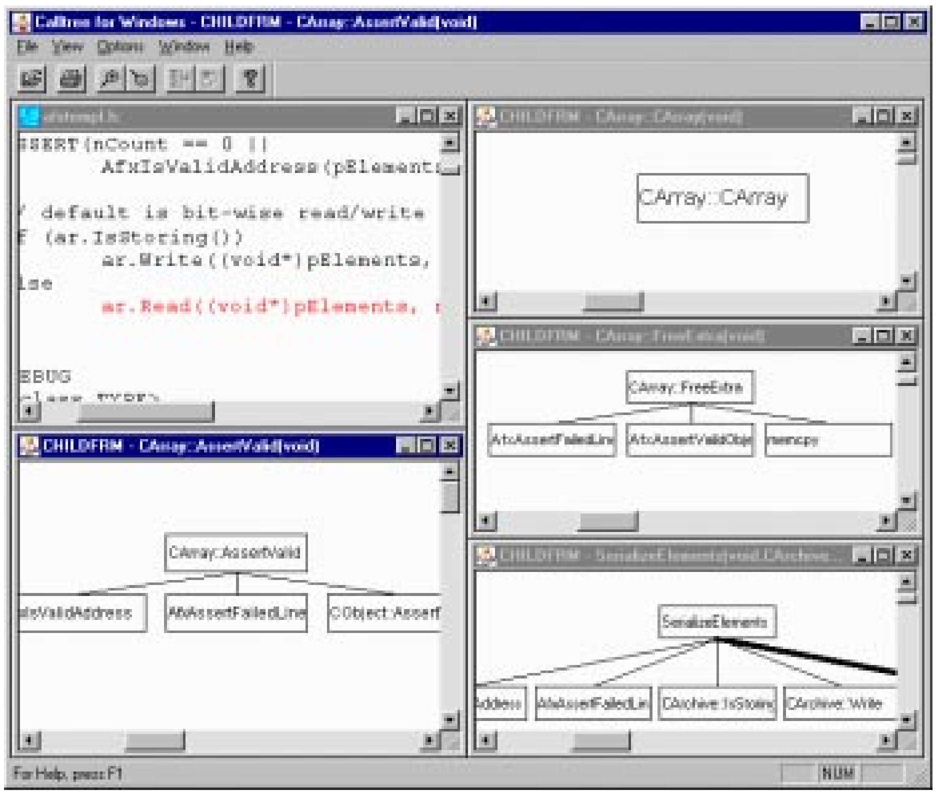


FIGURE 60 Tiled Windows in CallTree

7.7.3 Arrange Icons

When you have minimized windows, this option arranges them neatly at the bottom of the **CallTree** window.

7.7.4 Window List Box

This area of the pull-down menu lists all the open windows available in **CallTree**. The active window is indicated by a check mark. To activate a new window, especially if the windows are fully overlapping, select it from this list.

7.8 Options Menu

This menu brings up a dialog box from which several display options are available.

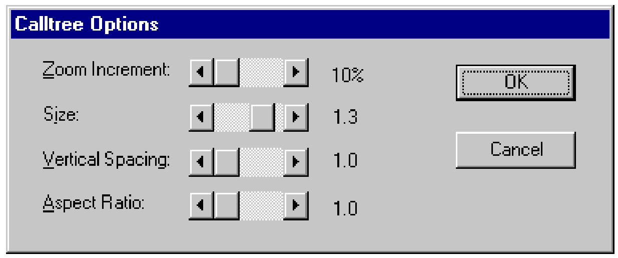


FIGURE 61 CallTree Options Dialog Box

This dialog box allows you to choose the magnification step used for the **Zoom In** and **Zoom Out** commands, the shape and size of the digraph's nodes, and the colors of the digraph's edges.

Zoom Increment This sets the magnification interval for the **Zoom In** and **Zoom Out** options. The default setting is .1 meaning a 10% reduction or enlargement in scale each time these buttons are used. To change the setting, move the slider left or right. Each 0.1 represents 10%, so if you slide the rule to .3, for example, the reduction and enlargement is 30% each time.

Vertical Spacing This alters the vertical distance between members of callpairs.

Aspect Ratio This alters the distance between and the width of the boxes.

OK If you click on the **OK** button, all the current settings in the **Options** window are applied to the calltree.

Cancel If you click on the **Cancel** button, any changes you have made since opening the **Options** window are discarded.

APPENDIX A

C/C++ Instrumentor Engine Database Files

This file lists examples of **WinIC9**'s output files. This appendix applies to all editions of **Coverage for Windows**.

A.1 Instrumentation Database Definitions

This section outlines the files that are used in the instrumentation database stored in the tcat_db directory. This information is used throughout **Coverage for Windows**.

A.1.1 d_graph Files

The digraphs for each function are put into files which are named with the same basename as the file from which they originated, with any filename suffix stripped off.

The format of each d_graph file is a set of blank delimited (white space delimited) lines composed as follows:

```
tail  head  edge  fun_id  type  filename
lbeg  lend  byte_beg  byte_end  string
result  [byte1  byte2]
```

where the fields have the following meanings:

tail	The tail node number (string)
head	The head node number (string)
edge	The **ic9** assigned edge number (string), also known as the seg ID
fun_id	The number of the function, whose name is found in the **mdf** file
type	The type of statement which gave rise to the edge
filename	The filename where the original text of the program was found
lbeg	The beginning line number, in the named file, where the tail node is found
lend	The ending line number, in the named file, where the head node is found
byte_beg	The beginning byte number, in the named file, where the tail node is found
byte_end	The ending byte number, in the named file, where the head node is found
string	The text string associated with the logical expression that headed the segment
result	The result corresponding to this edge, e.g. **T** or **F** or **36** (for switch outcome)
[byte1 byte2]	Currently “0 0”; reserved for expansion

A sample *d_graph* file is listed in Section A.2.1

A.1.2 c_graph Files

The calltrees for each processed file are put into files which are named with the same basename as the file from which they originated, with any filename suffix stripped off.

The format of each **c_graph** file is as a set of blank delimited (white space delimited) lines composed as follows:

```
file.caller callee callpair_id module_id
   source_file line 0 0 Segment_id
```

where the fields have the following meanings:

file.caller	The file name (given as a prefix up to the rightmost "." in the token, and the name of the calling function (the "caller")
callee	The name of the called function
callpair_id	The assigned identification number of the call pair
module_id	The assigned identification number of the module. This number points into the **mdf** file
source_file	The name of the source file that gave rise to the call pair
line	The line number of the source file where the call pair exists
0 0	These two fields are pre-set to be "0 0"
segment_id	(Reserved for future releases)

An example *c_graph* file is given in Section A.2.2.

A.1.3 Module Definition Files (mdf)

The *mdf* file contains basic information about the location of text fragments for every segment and every call pair in all processed files.

The ***mdf*** file has the following format:

```
project-name #segs #CPs [#rels]
file.name.function_id type #segs #CPs
[#rels]
file.name.function_id type #segs #CPs
[#rels]
file.name.function_id type #segs #CPs
[#rels]
...
```

where the first line identifies:

project-name	This is the name of the “project” from which the data is taken.
#segs	This is the total number of segments in the project.
#CPs	This is the total number of call pairs in the project.

The subsequent lines' fields have the following meanings:

file.name	This token contains, first, the name of the file in which the function name was found, and second, after the rightmost “.”, the name of the function.
function_id	This is the unique numeric identifier for that function, as found in the filename, which prefixes the function name.
type	This is the type of function that was processed according to the key: 84 = static function; 111 = member function. Note: These numbers are implementation specific. Additional function types and different codes will be added in the future. At present this function type information is not used.
#segs	This is the number of segments in the function.
#CPs	This is the number of call pairs in the function.

An example *mdf* file is given in Section A.2.3.

A.1.4 Trace Files and Archive Files

The format described is the Type 3.0 variation that produces trace files that are “self describing" and need no other files to be processed correctly. The assumption is that the assignment of numbers to modules is done by a runtime lookup of each module's name.

The format for an Archive File is identical except that the records are arranged in the “natural” order.

The trace file format is universal for all types of runtimes used and for either trace files or archive files. The record definitions have the following meanings:

#Format number Trace file Format Type Record

Defines the type of the current trace file. This line MUST appear as the first line of the trace file:

#Format 3.0

If it does not then this trace file is assumed to be one using a prior set of definitions.

 # comment Comment Line Record

The entire line is treated as a comment. Any blank line in the trace file is ignored. Tabs and extra spaces are treated as singleton blanks (i.e. as white space). The trace file line can be any length (subject to system constraints).

@ date Creation Date Record

This is the time and date stamp for the trace file, output taken from date.

p filename F X The Project

The first argument is project name. The first number represents the number of functions.

n"M N nsegments Module Definition Record. The module name M has been entered, and it has been assigned run-time identification number N for the duration of this trace file. The module has *nsegments* segments and *ncallpairs* call pairs. The function name is listed with the pathname and file name preceding it.

This line is written out only the first time that the module was executed in the current test. (Second instances of this record can be ignored by the coverage analyzer.)

c "N M [ntimes]" Call Pair Hit Record

Call pair M in module N has been hit [*ntimes times*]. This record is used to support S1 coverage measurements.

In an archive file the *ntimes* show the total number of times this call pair was hit. If a call pair was not hit, the record need not appear for that segment.

s "N M [ntimes]" Logical Segment Hit Record. Segment M in module N has been hit [*ntimes times*]. This record is used to support C1 coverage measurements, and also is used to support S0 coverage measurements.

In an archive file the *ntimes* show the total number of times this segment was hit. If a segment was not hit the record need not appear for that segment.

A sample trace file is listed in Section A.2.4.

A.2 Example Instrumentation Database Files

Here are some examples of database files:

A.2.1 d_graph File

This is a typical *d_graph* file:

```
0 1 1 0 0 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 31 0 0 0 (1) 0 266240 2307
0 1 1 1 0 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 51 0 0 0 (1) 0 0 0
0 1 1 2 0 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 68 0 0 0 (1) 0 0 0
1 2 2 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 69 0 0 0 (!AfxOleInit()) 1 0 0
1 2 3 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 79 0 0 0 (!AfxOleInit()) 0 0 0
2 3 4 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 109 0 0 0 (!pMainFrame->Loa-Frame(2)) 1 0 0
2 3 5 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 110 0 0 0 (!pMainFrame->LoadFrame(2)) 0 0 0
3 4 6 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 118 0 0 0 (RunEmbedded()||RunAutomated()) 1 0 0
3 4 7 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 125 0 0 0 (RunEmbedded()||RunAutomated()) 0 0 0
4 5 8 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 129 0 0 0 (m_lpCmdLine[0]=='\0') 1 0 0
4 5 9 2 1 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 134 0 0 0(m_lpCmdLine[0]=='\0') 0 0 0
0 1 1 3 0 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 173 0 0 0 (1) 0 0 0
0 1 1 4 0 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 177 0 0 0 (1) 0 0 0
0 1 1 5 0 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 182 0 0 0 (1) 0 0 0
0 1 1 6 0 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 191 0 0 0 (1) 0 0 0c_graph File
```

A.2.2 c_graph File

This is a typical *c_graph* file:

```
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) AfxOleInit(int) 1 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 68 0 0 1
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) AfxMessageBox(int,Cchar*,Uint,Uint) 2 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 70 0 0 2
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) SetDialogBkColor(void,CWinApp&,Ulong,Ulong) 3 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 79 0 0 3
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) LoadStdProfileSettings(void,CWinApp&) 4 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 80 0 0 3
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) AddDocTemplate(void,CWinApp&,CDocTemplate*) 5 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 93 0 0 3
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) COleTemplateServer::ConnectTemplate(void,CGUID&,CDocTemplate*,int) 6 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 98 0 0 3
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) COleTemplateServer::RegisterAll(int) 7 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 102 0 0 3
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) EnableShellOpen(void,CWinApp&) 8 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 113 0 0 5
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) RegisterShellFileTypes(void,CWinApp&) 9 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 114 0 0 5
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) RunEmbedded(int,CWinApp&) 10 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 117 0 0 5
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) RunAutomated(int,CWinApp&) 11 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 117 0 0 5
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) COleTemplateServer::UpdateRegistry (void,OLE_ APPTYPE,Cchar**,Cchar**) 12 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 125 0 0 7
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) OnFileNew(void,CWinApp&) 13 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 131 0 0 8
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) OpenDocumentFile(CDocument*,CWinApp&,Cchar*) 14 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 136 0 0 9
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) CWnd::DragAcceptFiles(void,int) 15 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 139 0 0 9
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) ShowWindow(int,CHWND__*,int) 16 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 141 0 0 9
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) UpdateWindow(void,CHWND__*) 17 2 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 142 0 0 9
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CAboutDlg::DoDataExchange(void,CDataExchange*) CDialog::DoDataExchange(void,CDataExchange*) 1 4 C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CPP 177 0 0 1
```

A.2.3 mdf File

This is a typical mdf file:

```
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::GetMessageMap(AFX_MSGMAP*) 0 100 1 0
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::{(void) 1 100 1 0
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) 2 100 9 17
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CAboutDlg::{(void) 3 100 1 0
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CAboutDlg::DoDataExchange(void,CDataExchange*) 4 100 1 1
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CAboutDlg::GetMessageMap(AFX_MSGMAP*) 5 100 1 0
C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::OnAppAbout(void) 6 100 1 0
```

A.2.4 Trace File and Archive File

This is a typical trace file or archive file:

```
#Format 3.0
# Profile for project 'SCRIBBLE':
p SCRIBBLE 7 6
n C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::GetMessageMap(AFX_MSGMAP*) 0 100 1 0
n C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::{(void) 1 100 1 0
n C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::InitInstance(int) 2 100 9 17
n C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CAboutDlg::{(void) 3 100 1 0
n C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CAboutDlg::DoDataExchange(void,CDataExchange*) 4 100 1 1
n C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CAboutDlg::GetMessageMap(AFX_MSGMAP*) 5 100 1 0
n C:\STW\TCAT\SAMPLES\SCRIBBLE\SCRIBBLE.CScribbleApp::OnAppAbout(void) 6 100 1 0
# End of profile for project 'SCRIBBLE'.
s 0 1 15458
s 1 1 1
s 2 1 1
s 2 3 1
s 2 5 1
s 2 7 1
s 2 8 1
c 2 1 1
c 2 3 1
c 2 4 1
c 2 5 1
c 2 6 1
c 2 7 1
c 2 8 1
c 2 9 1
c 2 10 1
c 2 11 1
c 2 12 1
c 2 13 1
c 2 15 1
c 2 16 1
c 2 17 1
s 3 1 1
s 4 1 2
c 4 1 2
s 5 1 88
s 6 1 1
```

cover9 —TCAT C/C++'s Coverage Analyzer

This section explains options for invoking and customizing the "cover9" coverage analyzer. This section applies to all editions of TCAT C/C++.

These are the options on how to invoke **cover9**. This command, used inside the **TCAT C/C++** graphical user interface, is used to produce a coverage report which, optionally, can report results in a Reference Listing. The Reference Listing report allows you to look up a segment in order to identify the actual unexecuted code, and plan new test cases.

C.1 Command Line Invocation

The complete syntax for calls to **cover9** is listed below. Items enclosed in [brackets] are to be included zero or more times.

```
cover9 [tracefile [tracefile]]
         [-a old-archive]
         [-b file]
         [-c]
         [-C1]
         [-d name [name]]
         [-DI deinst-file]
         [-DL]
         [-f new-archive]
         [-h | -h name [name]]
         [-html | -html filename]
         [-H]
         [-N]
         [-n]
         [-nl namefile]
         [-NH]
         [-m]
         [-l | -l name]
         [-p]
```

```
[-q]
[-r report]
[-S0]
[-S1]
[-s]
[-SU]
[-T [threshold]]
[-w width]]
```

C.2 Cover9 Switch Definitions

The options may be used to vary the processing and reports generated by **cover9**. The options are listed in alphabetical order.

[tracefile [tracefile]] These are the names of the trace files that you wish to process. If there are no trace files then **cover9** looks for data in the default trace file name **Trace.trc**.

If there are no names given, and **Trace.trc** is not present then an error message is issued.

If there are multiple trace files, each trace file is processed in the order presented.

Caution: *The list of trace files must be the first set of arguments. The list is ended by the first symbol that appears with a '-', i.e. by the first optional switch.*

-a *old-archive* *Old Archive File Name Switch.*You can include data from an old archive file in your reports. On the standard cumulative coverage report, this data will be included in the "Cumulative Summary" test results, but not under the column "Test". To test iteratively, progressing through a structured series of tests towards higher C1 values, each run of **cover** should include the cumulative archive file from the previous test.

If you do not include an archive file, the "Cumulative Summary" figures will be the same as those for "Test". Alternatively, if no -a option is given, the file Archive is used by default.

The **-a** option interacts with the other report options discussed below.

-b *file* *Banner File Name Switch.* This allows you to include specific text, taken from the first line of the file named *title* as a title for your reports. A maximum of 80 characters is allowed for titles.

-c *Cumulative Report Switch.* This option prints the Cumulative report only.

-C1 *Branch Coverage Reporting Switch.* Turns on reporting of C1 or branch coverage.

Note: *Unless at least one of* ***-C1****,* ***-S1****, or* ***-S0*** *is turned on, no coverage report will be generated.*

-d *name* *Module Name Delete Switch.* If this switch is present then the named modules, if found in the current execution, are deleted from the generated Archive file. Subsequently, **cover9** will never have heard about these names. This switch is useful in updating an extensive test record that would otherwise be lost due to the complexity of editing the Archive file.

-DI *deinst-file* *De-instrument Switch.* Allows the user to specify a list of modules that are to be excluded from coverage reporting. Only the list of module names found in the specified `deinst-file` is to be excluded from coverage reporting. The module names can be specified in any format. White space (such as tabs, spaces) is ignored. `deinst-file` is also the file where new modules that pass the coverage threshold value (see the **-T** switch) will be written.

-DL *De-instrument Module List Switch.* Allows the user to see which modules are excluded from coverage reporting. This switch is used along with the **-DI** switch. The list of excluded modules is printed at the end of the coverage report

-f *new-archive* *New Archive File Name Switch.* Newly accumulated test coverage data will be placed in this file. If you do not include a different name with this switch, the accumulated test data will be placed in the default name Archive.

Caution: Each time you run **cover9**, you will write over the contents of the Archive file unless you use the -f switch to direct the Archive file to another place. You may wish to remove the filename before starting a new test sequence.

-h | -h [name] Linear Histogram Report Switch (**-h**).

-html *[filename]* *HTML Switch.* If present, the current coverage report in html format will be generated. Normally the report is written to the file Coverage.htm (the default name), but you can rename the file with this switch. CAUTION: You will overwrite any file you name with this switch.

-l | -l [name] Logarithmic Histogram Report Switch **(-l**).

These two options produce two "histogram" reports that graph the frequency distribution of the segments exercised in a single module. The histograms provide a module-by-module analysis of testing coverage, combining current trace file data with archive date included through the -a option or using the default Archive file. If the optional name argument is present, then the corresponding histogram for only the named module is produced; otherwise, **cover9** produces histograms for all modules found. There can be multiple names in the argument if you want histograms of several modules. Also, the names can be mixed between linear and logarithmic histograms.

-H *Hit Report Switch.* Lists the segments that have been hit one or more times in current or past tests. This report analyzes the cumulative effect of the current trace file and any archive data included through the use of the **-a** option or using the default Archive file.

-m *Minimal Output Switch.* When present, **cover9** suppresses banner information, list of current options and trace file descriptions. The coverage report contains only the reports requested.

-N, **-n** *Not Hit Report Switch.* This option produces the "Not Hit" report which lists segments that have not been exercised. This report analyzes the cumulative effect of the current trace file and any archive data included through the use of the **-a** option or using the default Archive file.

-NH *Newly Hit Report Switch.* Shows the segments by module that were hit in the current execution that were not hit previously. Thus this gives the user an assessment of the value of the most-recently added test(s). This shows what the current test "gained". Output is the complement of the "Newly Missed" report.

-nl *namefile* *Name List Switch.* This switch specifies that only the list of module names found in the specified *namefile* file is to be reported on in the current coverage report.

Coverage on other module names that may appear in the archive or supplied trace files are ignored; however, the data is accumulated in the archive file.

The names used must be specified one name per line. White space (tabs, spaces, etc.) on the line is ignored.

The following reports are affected by the existence of a namefile:

- Cumulative Report
- Past Report
- Not Hit Report
- Hit Report
- Newly Hit Report
- Newly Missed Report.

The histogram outputs are not affected. There is a separate name mechanism that can be used to produce individual histogram reports.

-NM *Newly Missed Report Switch.* This option produces the Newly Missed report. Shows which segments, by module, hit in any prior test that were not hit in the current test. This shows what the current test "lost". This output is the complement of the Newly Hit report.

-p *Past Report Switch.* Print only the Past Test report; this option should be used in conjunction with the -a option when you want to analyze the overall performance of a set of past tests.

-q *Quiet Output Switch.* Suppress printout of current version and release information (this can be used to facilitate running **cover9** in batch mode).

-r *report* *Coverage Report File Name Switch.* Normally the report is written to the file Coverage (the default name), but you can rename the file with this switch. CAUTION: You will overwrite any file you name with this switch.

-S1 *Call-Pair Coverage Switch.* If present, the report will show call pair coverage.

-S0 *Module Coverage Switch.* If present, the report will show module coverage.

NOTE: Unless at least one of **-C1**, **-S1**, or **-S0** is turned on, no coverage report will be generated. However, not both **-S1** and **-S0** can be present; if they are then only **-S1** is assumed.

-s — *Sort Switch.* This option produces output reports with module names sorted alphabetically.

-SU — *Suppress Update Switch.* During processing, **cover9** will suppress updating of the archive file, either the default Archive or the file named by the **-f** switch. **cover9** will read the data in the archive file to form the basis for the "past test" information.

-T *threshold* — *Coverage Threshold Switch.* Threshold is a real number that specifies threshold value. Any module with a coverage percentage greater than or equal to this threshold value will be written to the de-instrumented file (see the **-DI** `deinst-file` switch). If no threshold is specified, then the default value of 85 percent is assumed.

-w *width* — *Report Width Switch.* Normally the reports generated by **cover9** are wide enough to accommodate module names up to 21 characters in length. The internal limit on name length is, however, 128 characters. You can use this switch to force **cover9** system to generate reports that are wide enough to accommodate the full 128 character module names.

The width factor is the number of additional characters to be added to the report. The default value is zero. Maximum width is 128 - 21 = 107. **WARNING**: Reports with high values for the -w option may contain long lines and may not be suitable for printing directly.

C.3 Error Processing

In case there is an error, **cover9** gives a response line (usage line) indicating the set of switches and options. This response is the same as the **-help** response.

Coverage Report Layout

This section shows you a great detail of detail about the current test you are analyzing, and about how the current test relates to the history of all tests you have run for this project.

The current test data is stored in the Trace File (Figure 62, Point 1), and the summary of all test data is stored in the Archive File (Figure 62, Point 2).

Typically when you run Cover you supply a Trace File and an Archive File and after you've analyzed the coverage in your Trace File you have the option to update your test coverage archive with the new test data.

To do this you press "Update Archive" (see Figure 62, Point 3) to update the Archive file so that it contains the data reflecting both the past test (in the old Archive File) and the current tests (in the Trace File).

D.1 Project Data

The basic report also shows the current project name (Figure 62, Point 4) and a summary of the basic facts that TCAT knows about this project.

As shown at (Figure 62, Point 5) you learn the total number of files involved in the project, and the total number of functions contained in those files.

D.2 Total Project Test Coverage Data

The Cover report shows you the total test coverage achived, measured for both C1 (branch) and S1 (call-pair) coverage.

This is presented for the current test -- from the Trace File -- and for all of the test data as reflected in the combination of the Trace File data and the Archive File data.

This is shown at (Figure 62, Point 6). The display shows:

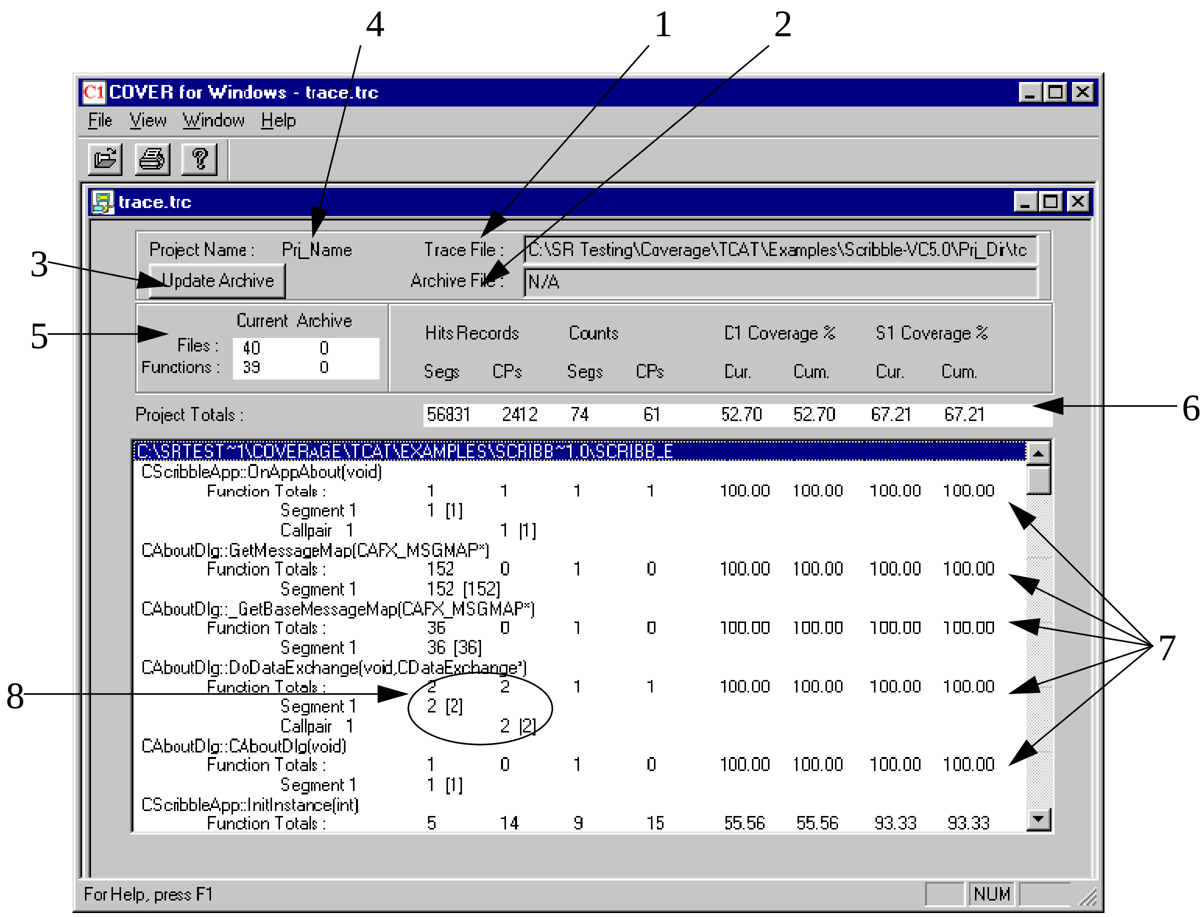


FIGURE 62 Coverage Report Analysis (TEST A)

- The total number of segment-hits in the current Trace File
- The total number of call-pair hits in the current Trace File
- The Total number of segments in the project
- The Total number of call-pairs in the project
- The achived branch (C1) percentage coverage in the current test
- The achived branch (C1) percentage coverage in all tests thusfar
- The achived callpair (S1) percentage coverage in the current test
- The achived callpair (S1) percentage coverage in all tests thusfar

These numbers give you a very good assessment of the coverage obtained for every test known.

D.3 Per-Function Test Coverage

The lower part of the Cover display is of variable format. If you click on the name of a file you see the expansion of the test coverage data for every function that is part of that file.

Click again on the display and the data collapses to show just the summary for that file.

For each function in the Function Totals line (Figure 62, Point 7):

- The total number of segment-hits for that function in the current Trace File
- The total number of call-pair hits for that function in the current Trace File
- The Total number of segments for that function
- The Total number of call-pairs for that function
- The achived branch (C1) percentage coverage for that function in the current test
- The achived branch (C1) percentage coverage for that function in all tests thusfar
- The achived callpair (S1) percentage coverage for that function in the current test
- The achived callpair (S1) percentage coverage for that function in all tests thusfar

D.4 In-Function Detailed Test Coverage

If you click on the Function Totals you will see an expansion that lists for each functionthe individual statistics for each segment and / or for each callpair, for as many as there are of these for that particular function.

The data (See Figure 62, Point 8) shows the number of times the particular segment or callpair was hit in the current test (in the number to the left), and the total number of times that segment or callpair was hit in all tests thusfar (the number in the [ ]'s on the right)

Note that if you click on a segment number of on a callpair you are taken directly to the source listing display and that particular part of the program that corresponds to that segment number of callpair number.

D.5 Interpreting Data From Multiple Tests

There is a great deal of data on the Cover display and if you have multipletest is sometimes can be hard to understand why things are the way they are.

For illustration we have shown five snapshots of Cover for the following set of tests.

Figure 62 shows the results of Test A with no Archive File

(See Figure 62)

Figure 63 shows the results of Test B with no Archive File

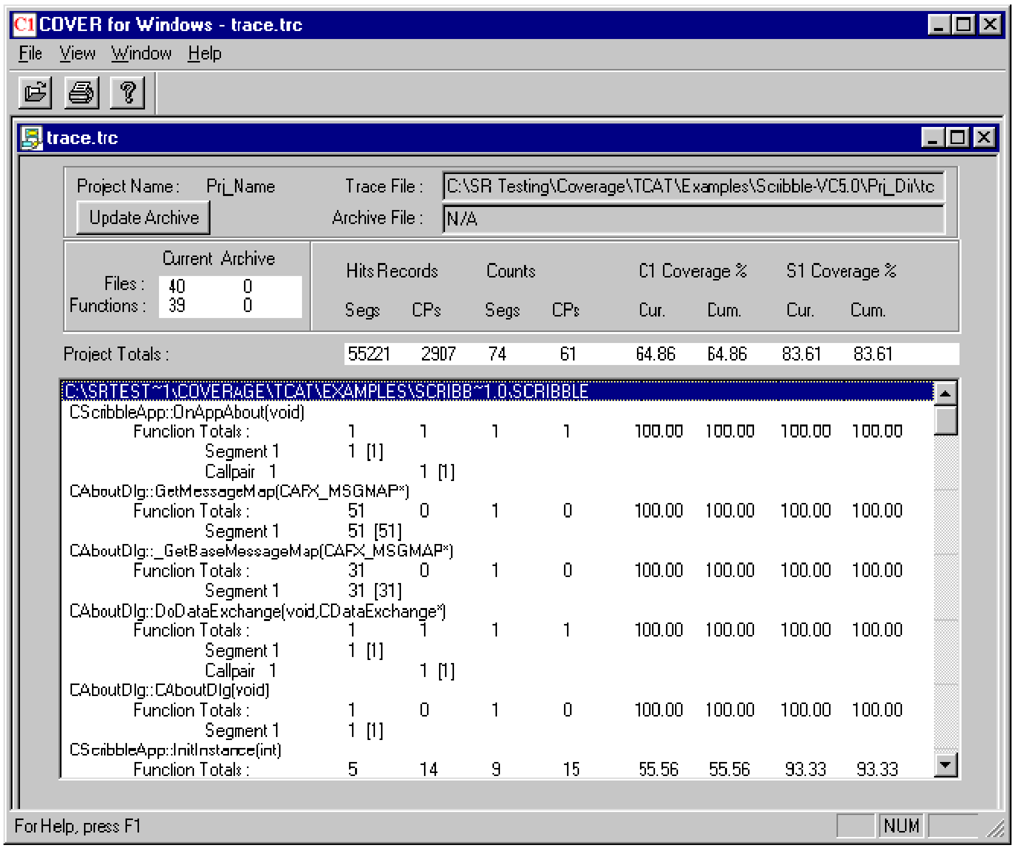


FIGURE 63 Coverage Report Analysis (TEST B)

Figure 64 shows the results of Test A + B with Test A's results as the Archive File and Test B's results as the current test.

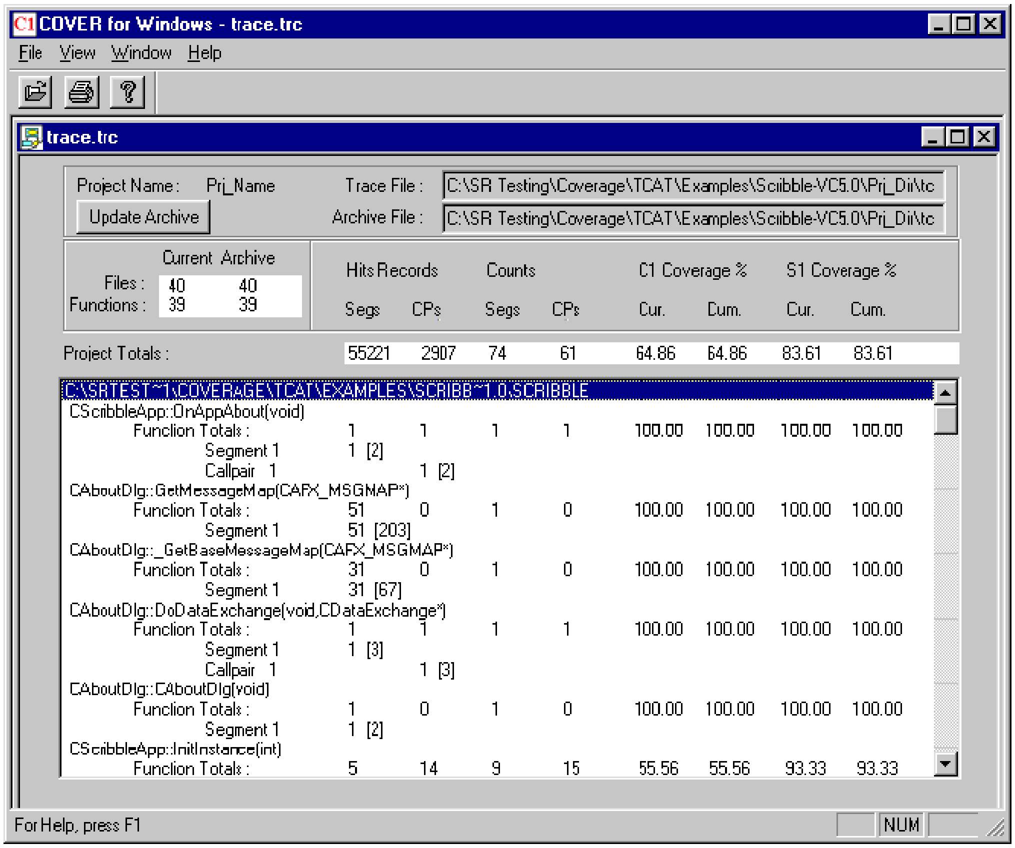


FIGURE 64 Coverage Report Analysis (TEST A+B)

Figure 65 shows the results of Test C with no Archive File.

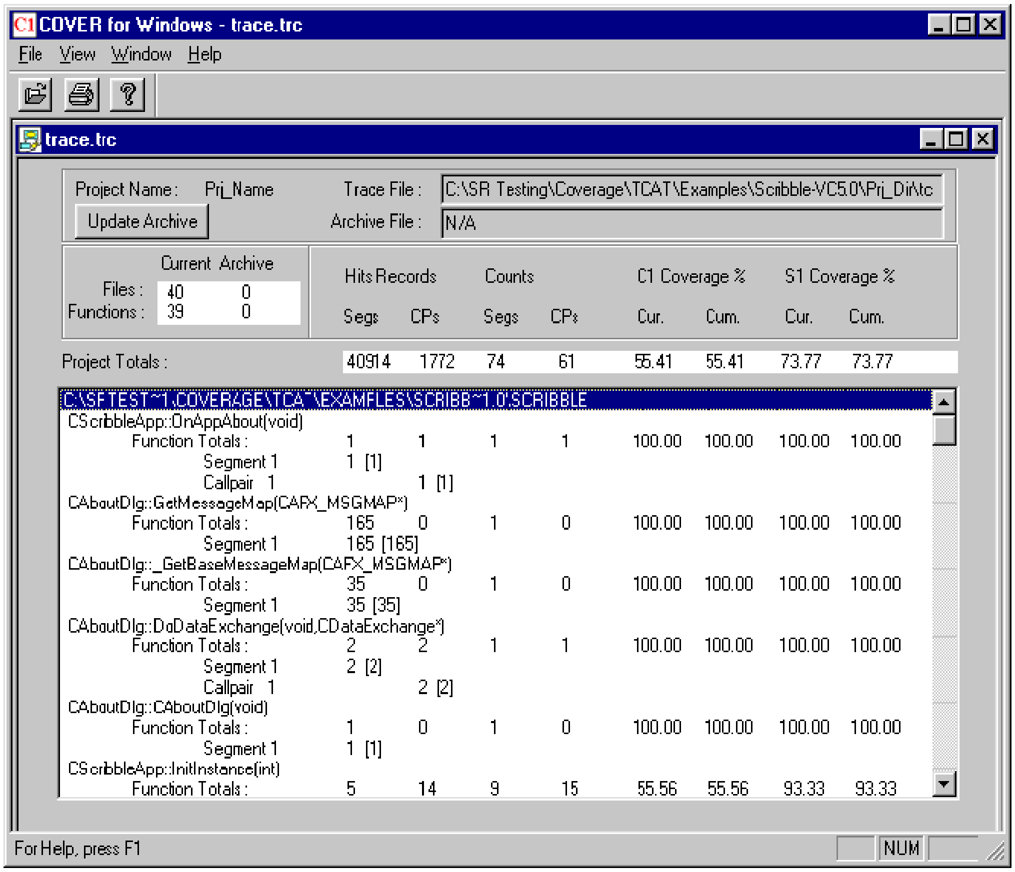


FIGURE 65 Coverage Report Analysis (TEST C)

Figure 66 shows the results of Test A + B + C using Test C as the current test and the archive file from Figure 64 (Test A + B) as the Archive file.

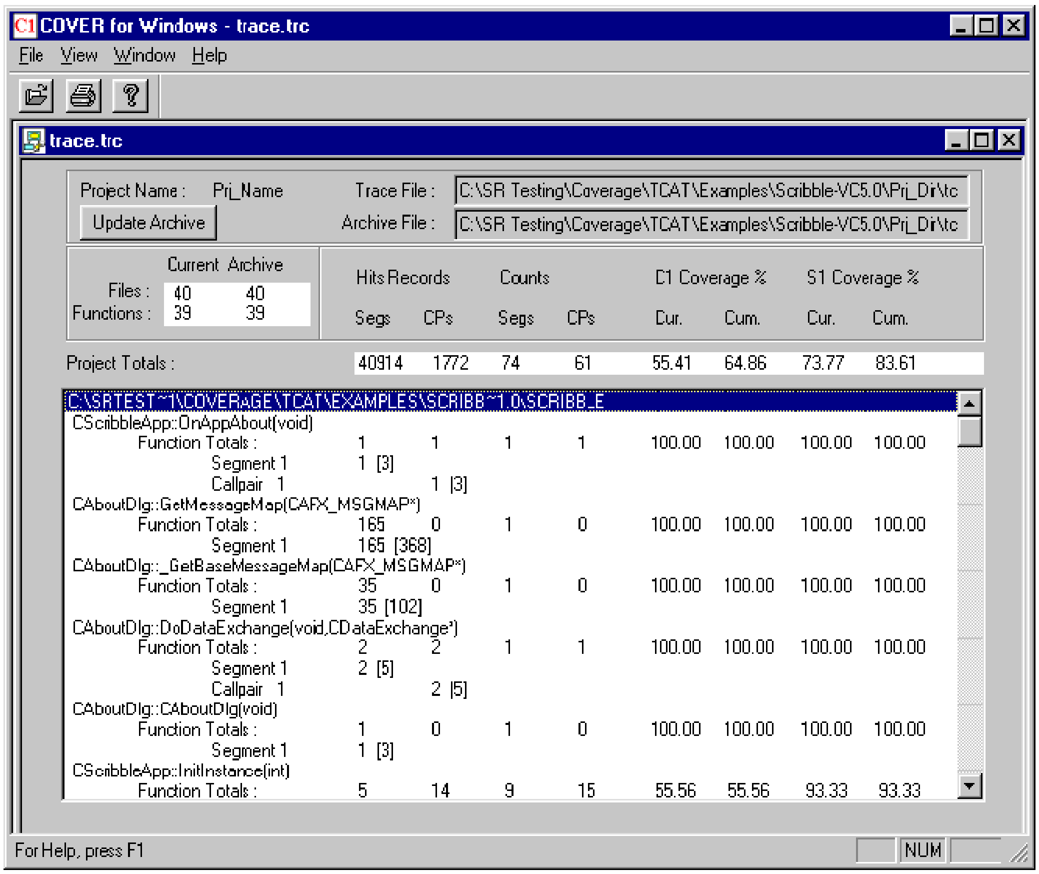


FIGURE 66 Coverage Report Analysis (TEST A+B+C)

It is worthwhile to spend a few minutes studying these results and to confirmthese facts about these three tests:

Test B is the best C1 test because it's results "mask" the two other tests.

You see this because Test A + B's cumulative results are no better than Test B by itself.

Test B is the also the best S1 test because it's results "mask" the two other tests.

You see this because Test A + B's cumulative results are no better than Test B by itself.

Remember when analyzing test coverage data that the C1 and S1 values for sets of tests grow in different ways, depending on what is done within the application.

As a result, the cumulative test coverage data values may exhibit some unusual and non-intuitive fluctuations.

Index

Symbols

Numerics

A

B

C

Q

R

S

T

U

V

W