

1

2

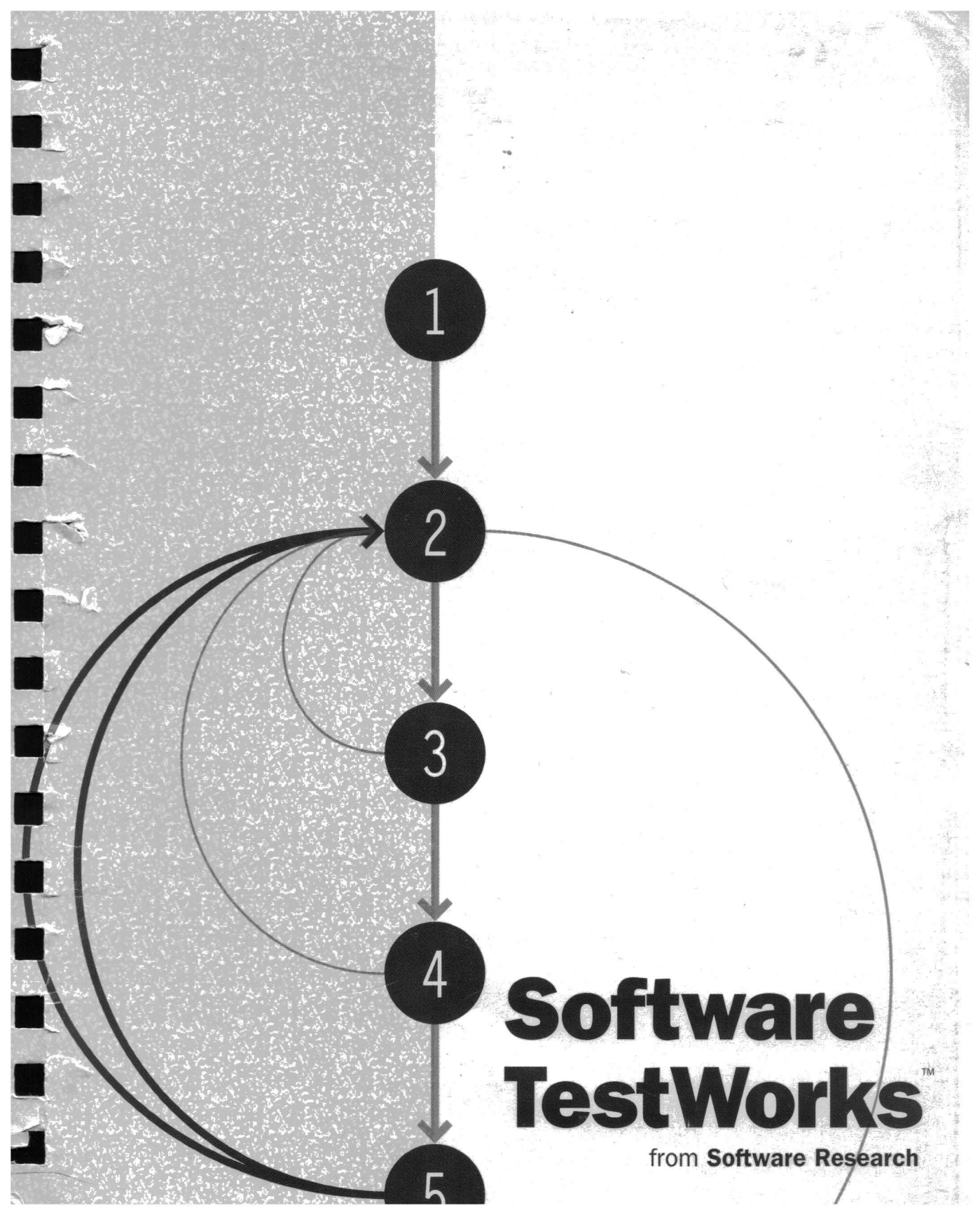
3

4

5

Software TestWorks™

from **Software Research**



Software Test Works

UNIX

STW/Coverage Tool Suite for C **(Book 1 of 2)**

TCAT: Test Coverage Analyzer

S-TCAT: System Test Coverage Analyzer



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

ALL RIGHTS RESERVED. No part of this document may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise without prior written consent of Software Research, Inc. While every precaution has been taken in the preparation of this document, Software Research, Inc. assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

TOOL TRADEMARKS: STW, CAPBAK, SMARTS, EXDIFF, TCAT, S-TCAT, TCAT-PATH, T-SCOPE, and TDGEN are trademarks of Software Research, Inc. METRIC is a trademark of SET Laboratories, Inc. and Software Research, Inc. and STATIC is a trademark of Software Research, Inc. and Gimpel Software.

Copyright © 1995 by Software Research, Inc
(Last Update July 17, 1995)

Table of Contents

PART I: TCAT USER'S GUIDE

CHAPTER 1: TCAT Overview	1
1.1 The QA Problem	1
1.2 The Solution	1
1.3 SR's Solution	2
1.4 Testing and TCAT	3
1.5 Software Test Methods	6
Manual Analysis	6
Static Analysis	6
Dynamic Analysis	6
1.6 Single- and Multiple-Module Testing	7
Bottom-Up	8
Top-Down	8
1.7 TCAT's Cost Benefits	8
Improved Error Detection	8
Earlier Error Detection	9
More Efficient Testing	10
Minimal Test Set	11
Assessment of Progress	11
CHAPTER 2: Quick Start	13
2.1 Recommendations	13
2.1.1 STEP 1: Starting Up TCAT	14
2.1.2 STEP 2: Invoking TCAT	16
2.1.3 STEP 3: Opening the Instrument Window	18

2.1.4	STEP 4: Choosing a Target Program	20
2.1.5	STEP 5: Running the Preprocessor	22
2.1.6	STEP 6: Instrumenting the Application	24
2.1.7	STEP 7: Opening the Execute Window	26
2.1.8	STEP 8: Compiling	28
2.1.9	STEP 9: Choosing a Runtime Version.....	30
2.1.10	STEP 10: Linking the Application	32
2.1.11	STEP 11: Running the Application - Part 1.....	34
2.1.12	STEP 12: Running the Application - Part 2	36
2.1.13	STEP 13: Opening the Analyze Window	38
2.1.14	STEP 14: Choosing a Trace File	40
2.1.15	STEP 15: Choosing Coverage Reports.....	42
2.1.16	STEP 16: Viewing the Coverage Reports.....	44
2.1.17	STEP 17: Selecting a Digraph of a Module	46
2.1.18	STEP 18: Viewing a Logical Branch's Source Code	48
2.1.19	STEP 19: Sign Off and Cleanup	50
2.2	Summary	52
 CHAPTER 3: System Introduction		53
3.1	Overview of TCAT	53
3.2	How to Use TCAT	53
3.2.1	Preprocessing Source Code.....	57
3.2.2	Instrument Program Code	57
3.2.3	Compile and Link Code	73
3.2.4	Execute Program and Generate Trace File	73
3.2.5	Generate Coverage Reports	73
3.3	Conclusion	87
 CHAPTER 4: GUI Operation.....		89
4.1	User Interface.....	89
4.1.1	File Selection Box	89
4.1.2	Help Boxes	91
4.1.3	Message Boxes	92
4.1.4	Option Menus	93
4.2	Invoking TCAT.....	96
4.2.1	Selecting Main Window Options.....	97
4.2.2	Exiting the Main Window	98
4.3	Instrumenting.....	98
4.3.1	Selecting the Application Name.....	99
4.3.2	Setting Options	100

	Preprocessing Option Menu	100
	Preprocessor output suffix	101
	Preprocessor command.....	101
	Preprocessor options	101
	Instrumentor command.....	101
	Instrumentor options	101
4.3.3	Preprocessing Your Program.....	102
	Preprocessing Results	103
4.3.4	Instrumenting Your Program.....	104
	Instrumenting Results	104
4.3.5	Exiting the Instrument Window	105
4.4	Running Your Program	106
4.4.1	Invoking the Execute Window	106
4.4.2	Setting Options	107
4.4.3	Compiling the Instrumented Program.....	108
	Compilation Results	108
4.4.4	Selecting a Runtime Object Module	108
4.4.5	Linking	109
	Linking Results	109
4.4.6	Running Your Application	110
	Running Results	110
4.4.7	Exiting the Execute Window	111
4.5	Using make Files	111
4.5.1	Preprocessing, Instrumenting, Compiling.....	111
4.5.2	Linking Object Modules	112
4.5.3	Example make Files	113
4.5.4	Running Your Make File	115
4.6	Obtaining Coverage Reports.....	116
4.6.1	Invoking the Analyze Window.....	117
4.6.2	Selecting the Trace File Name	118
4.6.3	Selecting Reports	118
4.6.4	Selecting Coverage Analyzer Options.....	121
4.6.5	Running the Coverage Analyzer.....	122
4.6.6	Looking at Coverage Reports.....	122
4.6.7	Exiting the Analyze Window	123
CHAPTER 5:	GUI Reference.....	125
5.1	TCAT Menus	125
5.2	Main Window.....	126
5.2.1	System Pull-Down Menu	127
5.2.2	Help Button.....	127

5.3	Instrument Window	129
5.3.1	File Pull-Down Menu	130
5.3.2	Action Pull-Down Menu	131
5.3.3	Help Button	131
5.3.4	Preprocessing Option Menu	132
5.3.5	Preprocessor output suffix Specification Region	132
5.3.6	Preprocessor command Specification Region.....	132
5.3.7	Preprocessor options Specification Region.....	132
5.3.8	Instrumentor command Specification Region.....	132
5.3.9	Instrumentor options	132
	Recognize _exit as keyword Button	132
	Do not recognize _exit as keyword Button.....	133
	Do not instrument functions in file Button.....	133
	Specify maximum file name length Butto	133
	Specify maximum function name length Button.....	134
5.4	Execute Window	135
5.4.1	File Pull-Down Menu	136
5.4.2	Action Pull-Down Menu	137
5.4.3	Help Button	138
5.4.4	Compiler command Specification Region.....	139
5.4.5	Compiler options Specification Region.....	139
5.4.6	Linker Command Specification Region	139
5.4.7	Linker options Specification Region	139
5.4.8	Make command Specification Region	139
5.4.9	Make file name Specification Region	139
5.4.10	Application name Specification Region	140
5.4.11	Application argument.....	140
5.5	Analyze Window	141
5.5.1	File Pull-Down Menu	142
5.5.2	Action Pull-Down Menu	143
5.5.3	Help Button.....	144
5.5.4	Past tests Check Button	145
5.5.5	Cumulative tests Check Button	146
5.5.6	Hit Check Button.....	147
5.5.7	Not Hit Check Button	148
5.5.8	Newly Hit Check Button	149
5.5.9	Newly missed Check Button	150
5.5.10	Log histogram Check Button	151
5.5.11	Linear histogram Check Button	152
5.5.12	Reference listing Check Button	153
5.5.13	Do not report function in file Check Button.....	154
5.5.14	Generate list of functions with C1> Check Button	154
5.5.15	Generate list of functions not included in report Check Button.....	154

5.5.16	Do not update archive file Check Button	155
5.5.17	Old Archive name Check Button	155
5.5.18	New Archive name Check Button	155
5.5.19	Rename the report file to: Check Button	155
5.5.20	Change the report width to: Check Button	155
5.5.21	Sort report by module name Check Button	156

CHAPTER 6: Command-Line Activation..... 157

6.1	Command Line Usage	157
6.2	'Xtcat' Command	157
	Options and Parameters:	157
6.3	ic Instrumentor Command	157
	Options and Parameters:	158
6.3.1	File Summary	161
6.3.2	Instrumentation Directive	162
	Application of Directive	162
	Proper Directive Placement	162
	Improper Directive Placement	166
	Additional Notes	166
6.4	cover Command	167
	Options and Parameters	167
6.4.1	File Summary	173
6.4.2	Trace File Argument	173
6.4.3	Archive Files	173
6.5	'mkarchive' Utility	174
6.6	Command Summary	174
6.6.1	Instrumentation, Compilation and Linking	174
	Stand-Alone Files	175
	Systems With make Files	175
	make Files With cc Called In Directives	175
	A System Which Does Not Use make File	176
6.6.2	Program Execution	176
6.6.3	Coverage Analysis	176

CHAPTER 7: Runtime Features

7.1	Runtime Descriptions	177
7.2	Special Runtimes	178

CHAPTER 8: Customizing TCAT	179
 PART II: S-TCAT	
 CHAPTER 9: Introduction	 183
9.1 Audience	183
9.2 Purpose	183
9.3 Manual Organization	184
 CHAPTER 10: Overview	 185
10.1 Why System Test Coverage Analysis?.....	185
10.2 QA Problems Addressed	185
10.3 Cost Benefit Analysis.....	186
10.3.1 Improved Error Detection	186
10.3.2 Earlier Error Detection	187
10.3.3 More Efficient Testing	188
10.3.4 Minimal Test Set	188
10.3.5 Assessment of Progress.....	188
10.4 Software Test Methods.....	189
10.4.1 Manual Inspection	189
10.4.2 Dynamic Analysis	189
10.5 Multiple-Module Testing.....	190
10.6 Hierarchy of Coverage Metrics.....	190
10.7 S1 Measure.....	191
10.8 How Does S1 Relate to C1?	191
10.9 Advanced Coverage Metrics	192
 CHAPTER 11: Instrumentation	 195
11.1 Overview.....	195
11.2 Instrumentation	195
11.2.1 The Instrumentor	196
11.2.2 Excluding Function Calls from Instrumentation	198
11.3 DOS Instrumentation.....	200
11.4 UNIX Instrumentation.....	200
11.4.1 Instrumenting With 'make' Files.....	200
11.4.2 Example 'make' Files.....	202

11.5	File Summary	209
11.6	Embedded Systems	210
CHAPTER 12: Compiling, Linking and Executing		211
12.1	Runtime Descriptions	211
	crun0 - Raw Tracefile ("quiet" runtime)	212
	crun1 - Standard Tracefile	212
	MS-DOS Runtimes	212
12.2	Special Runtimes (for UNIX only).....	213
	crun2 - In-Place Reductio	213
	crun3 - Multiple Processes	213
	cruna - Multi-Tasking (or forking runtimes	214
	crunc - Cross Development	214
12.3	Executing the Instrumented Program.....	215
	Performance Considerations	215
CHAPTER 13: Coverage Reporting and Analysis.....		217
13.1	Producing Reports	219
13.1.1	Report Types	219
13.1.2	Trace File Argument	219
13.1.3	Archive Files	220
13.1.4	'scover' Syntax.....	220
13.2	'mksarchive' Utility	225
13.3	File Summary	227
CHAPTER 14: Menus		229
14.1	S-TCAT/C ASCII Menus	229
14.1.1	Invoking S-TCAT.....	229
14.1.2	S-TCAT Menu Tree.....	230
	Issuing Commands	230
	Displaying Current Parameter Settings	231
	S-TCAT Menu 'Stack'	231
14.1.3	MAIN Menu.....	231
14.1.4	ACTIONS Menu.....	232
	FILES Menu	232
14.1.5	OPTIONS Menu	233
14.1.6	Saving Changed Option Settings	233
14.1.7	Running System Command.....	234

14.2	S-TCAT Configuration File	234
14.2.1	Configuration File Syntax	234
14.2.2	Sample S-TCAT Configuration File.....	236
CHAPTER 15: Command Summary: MS-DOS, OS/2		237
15.1	Instrumentation, Compilation and Linking	237
15.1.1	Stand-Alone Files	237
15.1.2	Systems With 'make' Files.....	238
15.1.3	'make' With 'cl', 'msc'	238
15.1.4	Systems Without 'make' Files	238
15.1.5	Program Execution	239
15.2	Coverage Analysis	239
CHAPTER 16: Command Summary-UNIX		241
16.1	Instrumentation, Compilation and Linking	241
16.1.1	Stand-Alone Files	241
16.1.2	Systems With 'make' Files.....	241
16.1.3	'make' files with cc called in directives.....	242
16.1.4	A system which does not use 'make' files	242
16.2	Program Execution	242
16.3	Coverage Analysis	243
CHAPTER 17: Full S-TCAT Example		245
17.1	Introduction	245
17.2	Preprocess, Instrument, Compile and Link	249
17.3	Reference Listing	254
17.4	Instrumentation Statistics	259
17.5	Call-Pair Listing	260
17.6	Reading S-TCAT Reports	261
17.6.1	Cumulative Report	261
17.6.2	Past Report.....	263
17.6.3	Not Hit Report.....	263
17.6.4	Hit Report	265
17.6.5	Newly Hit Report.....	266
17.6.6	Newly Missed Report.....	266
17.6.7	Linear Histogram	267
17.6.8	Logarithmic Histogram.....	269
17.6.9	Reference Listing S1 Report	270

17.7	Summary	273
CHAPTER 18: Graphical User Interface (GUI) Tutorial		
18.1	Invocation.....	275
18.2	Using S-TCAT/C	276
18.2.1	Instrument.....	277
18.2.2	Execute.....	279
18.2.3	Analyze.....	283
CHAPTER 19: Testing Guidelines: S-TCAT/C.....		
CHAPTER 20: System Restrictions and Dependencies		
CHAPTER 21: References		
PART III: SOURCE-VIEWING UTILITIES		
CHAPTER 22: Xdigraph Utility.....		
22.1	Purpose	299
22.2	Xdigraph File Format.....	299
22.2.1	All digraph files:	299
22.2.2	Multiple digraph files:	299
22.3	Invoking Xdigraph	300
22.4	Xdigraph Main Window	302
22.4.1	File	302
22.4.2	Options	302
22.4.3	Zoom In	302
22.4.4	Zoom Out.....	303
22.4.5	View Source.....	303
22.4.6	Statistics	303
22.4.7	Print	303
22.4.8	Annotation	303
22.4.9	Help	303
22.5	File Pull-Down Menu	304
22.5.1	Load New Graph	304
22.5.2	Load New Module.....	304
22.5.3	Set Archive	305

22.5.4	Exit.....	305
22.5.5	Digraph File Message Box	306
22.5.6	Filter.....	306
22.5.7	Directories	306
22.5.8	Files	307
22.5.9	Selection.....	307
22.5.10	OK	307
22.5.11	Filter Button	307
22.5.12	Cancel	307
22.6	Options Window	308
22.6.1	Zoom Scale	308
22.6.2	Node Characteristics	308
	Shape.....	309
	Size	309
	Vertical Spacing	309
	Aspect ratio	309
22.6.3	Edge Characteristics	309
	Unhighlighted Edge	309
	Eccentricity	309
	Default Color	309
	Low-level Color	309
	Normal Color	309
	High-level Color	310
	Apply	310
	Reset	310
	Close	310
	Help	310
22.7	Zoom In/Zoom Out Window	311
22.8	View Source Window.....	312
22.9	Statistics Window	313
22.9.1	File Name	314
22.9.2	Node and Edge Count.....	314
22.9.3	Cyclomatic Number (Cyclomatic Complexity)	314
22.9.4	Average, Minimum and Maximum Path Lengths	314
22.9.5	Path Count by Iteration Groups	314
22.10	Print Window.....	315
22.10.1	Paper Size Information	315
	Top Margi	315
	Left Margin	315
	Page Width.....	316
	Bottom Margin.....	316
	Right Margin.....	316

Page Height	316
22.10.2 Enlargement Factors	316
Width/Height	316
22.10.3 Font Information	317
Font name/Font size	317
22.10.4 Print locator	317
To File	317
To Printer	317
22.11 Annotation Window	318
22.11.1 Threshold 1 & 2	318
22.11.2 None	319
22.11.3 Nhits	319
22.11.4 N%	319
22.11.5 Complexity	319
22.11.6 Ntokens	319
22.11.7 Nlines	319
22.11.8 User	319
22.11.9 Highlight	320
22.11.10 Path File	320
22.11.11 Apply	320
22.11.12 Reset	320
22.11.13 Close	320
22.11.14 Help	320
22.11.15 Colors	320
22.12 Quick Reference Guide to Xdigraph Annotations	322
CHAPTER 23: Xcalltree Utility	323
23.1 Calltree Defined	323
23.2 Xcalltree File Format	323
23.3 Invoking Xcalltree	324
23.4 Xcalltree Main Window	325
23.4.1 File	325
23.4.2 Options	325
23.4.3 Zoom In	326
23.4.4 Zoom Out	326
23.4.5 View Source	326
23.4.6 Statistics	326
23.4.7 Print	326
23.4.8 Annotation	326
23.4.9 Help	326
23.5 File Pull-Down Menu	327

23.5.1	Load New Graph	327
23.5.2	Load New Multi Graph.....	327
23.5.3	Set Archive	327
23.5.4	Exit.....	328
23.6	Calltree File Selection Dialog Box	329
23.6.1	Filter.....	329
23.6.2	Directories	329
23.6.3	Files.....	329
23.6.4	Selection.....	330
23.6.5	OK	330
23.6.6	Filter.....	330
23.6.7	Cancel	330
23.7	Option Window	331
23.7.1	Zoom Scale	331
23.7.2	Horizontal Spacing	331
23.7.3	Depth	331
23.7.4	Root Name	332
23.7.5	Edge Characteristics	333
	Edge Color	333
	Unhighlighted Edge	334
	Display Mode	334
23.7.6	Node Characteristics	334
	Size	334
	Aspect Ratio.....	334
	Default Color	334
	Low-level Color	334
	Normal Colo	334
	High-level Color.....	334
	Apply	335
23.8	Zoom In & Zoom Out Options	335
23.9	View Source Window.....	336
23.9.1	Description of Source Code Viewing	336
23.10	Statistics Window.....	337
23.10.1	Links	337
23.10.2	Call pairs.....	337
23.10.3	Modules/Depth	337
23.10.4	Recursive.....	338
23.11	Print Window.....	338
23.11.1	Paper Size Information	338
	Top Margin	338
	Left Margin	339
	Page Width	339

Bottom Margin	339
Right Margin	339
Page Height	339
23.11.2 Enlargement Factors	339
Width/Height	339
23.11.3 Font Information	340
Font name/Font size	340
23.11.4 Print locator	340
To File	340
To Printer	340
23.12 Annotation Window	341
23.12.1 Threshold 1 & Threshold 2	341
23.12.2 None	342
23.12.3 S0	342
23.12.4 Ninvokes	342
23.12.5 S1	342
23.12.6 C1	342
23.12.7 Cyclo.....	342
23.12.8 Nsegs	342
23.12.9 Npairs	342
23.12.10Nlines.....	342
23.12.11Ntokens.....	343
23.12.12Npaths	343
23.12.13User	343
23.12.14Connections.....	343
23.12.15Apply.....	343
23.12.16Reset	343
23.12.17Close	343
23.12.18Help	343
23.13 Quick Reference Guide to Xcalltree Annotations.....	345
 CHAPTER 24: Index of Terms	 347

NOTE: Documentation for TCAT-PATH and T-SCOPE, the accompanying products in the STW/COVERAGE tool set, is included in STW/COVERAGE/BOOK II.



List of Figures

FIGURE 1	STW/Coverage Dependency Chart	3
FIGURE 2	STW/Coverage System Chart.	5
FIGURE 3	Stages in Software Testing.	7
FIGURE 4	Cost Benefit Analysis	9
FIGURE 5	Increase in Cost-to-fix Throughout Life-cycle	10
FIGURE 6	Setting Up the Display (Initial Condition)	15
FIGURE 7	Invoking TCAT	17
FIGURE 8	Initializing the Instrument Window	19
FIGURE 9	Selecting a Target Program	21
FIGURE 10	Preprocessing Your Program.	23
FIGURE 11	Instrumenting Your Program	25
FIGURE 12	Initializing the Execute Window.	27
FIGURE 13	Compiling the Instrumented Program.	29
FIGURE 14	Selecting a Runtime Object Module	31
FIGURE 15	Linking Object Modules	33
FIGURE 16	Naming the Trace File	35
FIGURE 17	Running the Application.	37
FIGURE 18	Initializing the Analyze Window	39
FIGURE 19	Selecting a Trace File Name.	41
FIGURE 20	Selecting the Reference Listing File	43
FIGURE 21	Looking at Coverage Reports	45
FIGURE 22	Selecting a Module	47
FIGURE 23	Looking at Source Code	49
FIGURE 24	Completing a TCAT Session.	51
FIGURE 25	Sample C Program	57
FIGURE 26	Instrumented Program	63

FIGURE 27	Reference Listing	68
FIGURE 28	Instrumentation Statistics Sample	70
FIGURE 29	Segment Count Listing Sample	71
FIGURE 30	Directed Graph Listing	71
FIGURE 31	Directed Graph Display	72
FIGURE 32	Error Listing	73
FIGURE 33	Using a File Selection Dialog Box	90
FIGURE 34	Using the Help Dialog Box	92
FIGURE 35	Using a Dialog Box	93
FIGURE 36	Using an Option Menu	94
FIGURE 37	Using a Pull-down Menu	95
FIGURE 38	Invoking the Main Window	96
FIGURE 39	Invoking TCAT from the STW Tool Suite	97
FIGURE 40	Exiting the Main Window	98
FIGURE 41	Invoking the Instrument Window	99
FIGURE 42	Selecting the Program File Name	100
FIGURE 43	Exiting the Instrument Window	105
FIGURE 44	Invoking the Execute Window	106
FIGURE 45	Selecting the Runtime Object Module	109
FIGURE 46	Exiting the Execute Window	111
FIGURE 47	Uninstrumented UNIX Make File	114
FIGURE 48	Instrumented UNIX Make File	115
FIGURE 49	Obtaining Coverage Reports	117
FIGURE 50	Invoking the Analyze Window	117
FIGURE 51	Selecting the Trace File Name	118
FIGURE 52	Reference Listing File Selection	120
FIGURE 53	Looking at Coverage Reports	123
FIGURE 54	Exiting the Analyze Window	124
FIGURE 55	Main Window	126
FIGURE 56	System Pull-Down Menu	127
FIGURE 57	Help Window for the Main Window	128
FIGURE 58	Instrument Window	129
FIGURE 59	Set File Name Dialog Box	130
FIGURE 60	File Pull-Down Menu	130
FIGURE 61	Action Pull-Down Menu	131
FIGURE 62	Help Window for the Instrument Window	131

FIGURE 63	Execute Window	135
FIGURE 64	Set Runtime Obj. Module Selection Dialog Box	136
FIGURE 65	File Pull-Down Menu	137
FIGURE 66	Action Pull-Down Menu	138
FIGURE 67	Help Window for the Execute Window	138
FIGURE 68	Analyze Window	141
FIGURE 69	Set Input Trace File Name Selection Dialog Box	143
FIGURE 70	File Pull-Down Menu	143
FIGURE 71	Action Pull-Down Menu	144
FIGURE 72	Help Window for the Analyze Window	144
FIGURE 73	Past Report	145
FIGURE 74	Cumulative Report	146
FIGURE 75	Hit Report	147
FIGURE 76	Not Hit Report	148
FIGURE 77	Newly Hit Report	149
FIGURE 78	Newly Missed Report	150
FIGURE 79	Log Histogram Report	151
FIGURE 80	Linear Histogram Report	152
FIGURE 81	Reference Listing File Selection	153
FIGURE 82	Reference Listing Report	154
FIGURE 83	TCAT resource file.	180
FIGURE 84	Cost Benefit Analysis	186
FIGURE 85	Increase in Cost-to-fix Throughout Life-cycle	187
FIGURE 86	Stages in Software Testing.	190
FIGURE 87	Uninstrumented DOS Make File.	203
FIGURE 88	Instrumented DOS Make File	205
FIGURE 89	Uninstrumented UNIX Make File	207
FIGURE 90	Instrumented UNIX Make File.	208
FIGURE 91	System Components.	218
FIGURE 92	Sample "C" Program.	248
FIGURE 93	Instrumented Program Segment	253
FIGURE 94	Reference Listing	258
FIGURE 95	Instrumentation Statistics Sampl	259
FIGURE 96	Call-Pair Listing Example	260
FIGURE 97	Cumulative Coverage Report.	262
FIGURE 98	Not Hit Report	264

FIGURE 99	Hit Report	265
FIGURE 100	Newly Hit Report	266
FIGURE 101	Newly Missed Report.	267
FIGURE 102	Linear Histogram	268
FIGURE 103	Logarithmic Histogram	270
FIGURE 104	Reference Listing S1 Report.	273
FIGURE 105	Main Menu	275
FIGURE 106	STW/COV Invocation	276
FIGURE 107	Main Menu Help	277
FIGURE 108	Instrument Menu	278
FIGURE 109	Instrument Help Menu	278
FIGURE 110	File Pop-Up Menu	279
FIGURE 111	Execute Menu.	280
FIGURE 112	Execute Help Menu	281
FIGURE 113	Runtime Object Module Pop-Up Screen	282
FIGURE 114	Analyze Menu	283
FIGURE 115	Analyze Help Menu	283
FIGURE 116	Set Input Trace File Name Pop-Up Window.	286
FIGURE 117	Reference Listing Pop-Up Window	287
FIGURE 118	Past Test Report.	287
FIGURE 119	Cumulative Report.	287
FIGURE 120	Hit Report	288
FIGURE 121	Linear Histogram	288
FIGURE 122	Reference Listing (Part 1 of 2)	289
FIGURE 123	Reference Listing (Part 2 of 2)	289
FIGURE 124	Source Viewing Pop-Up Window	290
FIGURE 125	Source Viewing	291
FIGURE 126	Program edges as represented in a digraph.	301
FIGURE 127	Xdigraph Main Window	302
FIGURE 128	Digraph File Pull-Down Menu	304
FIGURE 129	Digraph File Message Box	306
FIGURE 130	Xdigraph Options Window	308
FIGURE 131	Zoom In feature illustrated	311
FIGURE 132	View Source Option Window	312
FIGURE 133	Statistics Option Window	313
FIGURE 134	Print Dialog Window	315

FIGURE 135	Annotation Thresholds Window	318
FIGURE 136	Sample Annotation for User Threshold.	321
FIGURE 137	Xcalltree Main Window	325
FIGURE 138	File Pull-Down Menu	327
FIGURE 139	Calltree File Selection Dialog Box	329
FIGURE 140	Option Window	331
FIGURE 141	Root Name Selection Window Example 1	332
FIGURE 142	Root Name Selection Window Example 2	333
FIGURE 143	Zoom In Option illustrated	335
FIGURE 144	View Source Window	336
FIGURE 145	Statistics Window	337
FIGURE 146	Print Window	338
FIGURE 147	Annotation Window.	341
FIGURE 148	“NOT DEFINED in reference file” message box.	344



USER'S GUIDE

TCAT

Test Coverage Analyzer

Ver 8.1



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

ALL RIGHTS RESERVED. No part of this document may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise without prior written consent of Software Research, Inc. While every precaution has been taken in the preparation of this document, Software Research, Inc. assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

TOOL TRADEMARKS: STW, CAPBAK, SMARTS, EXDIFF, TCAT, S-TCAT, TCAT-PATH, T-SCOPE, and TDGEN are trademarks of Software Research, Inc. METRIC is a trademark of SET Laboratories, Inc. and Software Research, Inc. and STATIC is a trademark of Software Research, Inc. and Gimpel Software.

CREDITS: Programming: Linh Dang, Tjiu Oh; Documentation: Deborah Steiner, Brian Kluepfel; Design: Rebekkah Graves.

TCAT Overview

This chapter explains the basic concepts behind coverage tools, and tells how they can save you time and money in your development process.

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 percent of the source is actually tested before release. With little more than half of the logic actually covered, many bugs go unnoticed until after release. Worse still, the actual percentage of logic covered is unknown to SQA management, making any informed management decisions impossible.

Questions such as when to stop testing or how much more testing is required are not answered 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 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. Cases testing the same logic clutter test suites and take the place of other cases which would test previously unexplored logic. Often testers are unsure of the 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. To ensure a reliable and solid software product, 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 cover, in some way, all the possible situations in which the program will be used.

Although a worthy goal, imagining every possible use, as well as developing test data and running them, 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: *STW/Coverage*. *STW/Coverage* ensures tests are more diverse than those which are chosen by reference to functional specification alone or based on a programmer's intuition. *STW/Coverage* ensures tests are as complete as possible by measuring against a range of high quality test metrics:

- C1, or branch/segment coverage, measures module testing at the unit or module testing level; it accesses the completeness of individual modules or small groups of module testing.
- S1, or call-pair coverage, measures all the interfaces of a complex system to be exercised.
- Ct, or equivalence class coverage, measures the number of times each path or path class in a module is exercised.

With the three test metrics, *STW/Coverage* ensures tests are as complete as possible. *STW/Coverage* includes the following products:

- *TCAT* does coverage at the logical branch (or segment) level and the call-graph level. It employs the C1 metric. You can choose to test a single module, multiple modules or the entire program using the C1 metric.
- *S-TCAT* does coverage at the call-pair level. It employs the S1 metric. After individual modules have been tested, you can test all the interfaces of the system using the S1 metric.
- *TCAT-PATH* does coverage at the logical path level. It employs the Ct measure. It can easily be programmed to include or to exclude the program's modules from analysis. This allows you to emphasize certain critical modules, once these are identified. *TCAT-PATH* allows you to extract and display the logical conditions that will cause that particular path to be exercised. Based on these conditions, you can design new test suites to exercise the path.
- *T-SCOPE* provides dynamic visualization of test attainment during unit testing and system integration. It is a companion tool for *TCAT*, *S-TCAT* and *TCAT-PATH*. While these tools report the status of modules after-the-fact, *T-SCOPE* visually demonstrates such things as segments and call-pairs hit or not hit while it is happening.

TCAT for the C language is the focus of this manual. For complete information on use of the other *STW/Coverage* products, please consult the proper User Manuals.

Below is a *STW/Coverage* flow chart. Boxes with darkened backgrounds represent the main components of *STW/Coverage*.

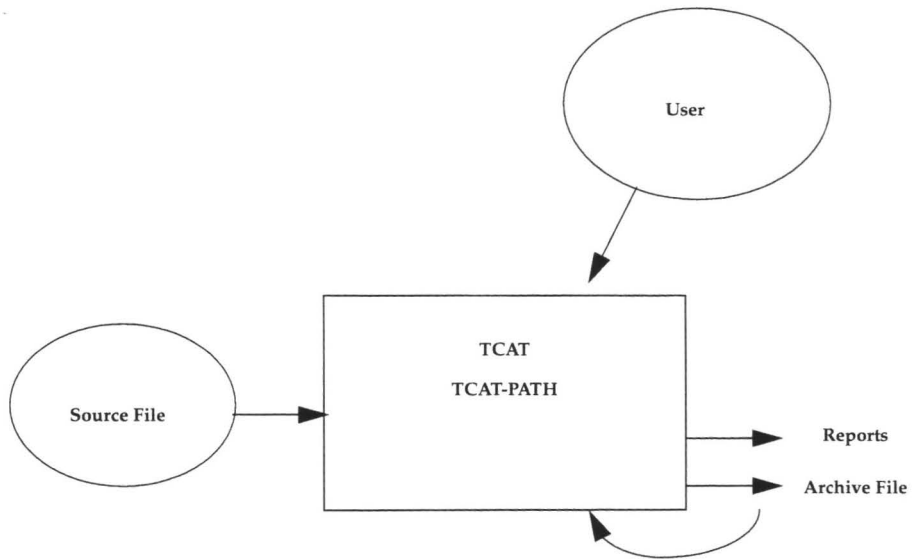


FIGURE 1 STW/Coverage Dependency Chart
1.4 **Testing and TCAT**

TCAT takes your program and automatically instruments it. During instrumentation, *TCAT* 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 report.
- A summary of past coverage runs.
- Current and cumulative coverage statistics.
- A list of logical branches that have been hit.
- Bar charts of the frequency of execution of each logical branch.

You should try to obtain >85 percent coverage. If the reports indicate that you have less than 85 percent coverage (the recommended amount), you can identify unexercised logical branches by looking at the entire reference listing report, or you can look at the reference listing code for a particular logical branch. When you identify the troubled areas, you can then create new test cases and re-execute the program.

TCAT can help you reach your goal: creating the most extensive test cases possible.

The diagram in the following figure illustrates the *TCAT* process. You should study this diagram carefully so that you see the natural structure and rhythm of *TCAT* use.

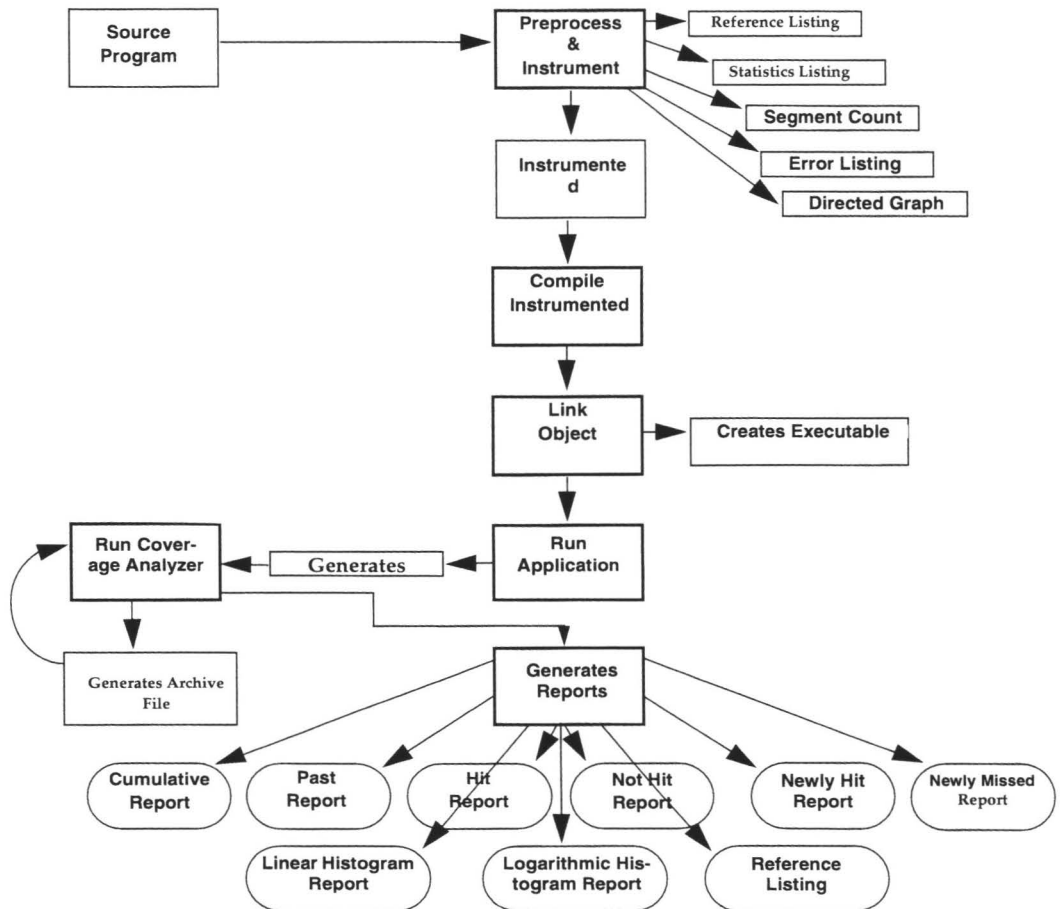


FIGURE 2 STW/Coverage System Chart

1.5 Software Test Methods

Coverage analysis as implemented through *TCAT* is a powerful testing technique which can save you much money and time, in addition to greatly improving software quality. Plainly, 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, which are usually performed in their natural sequence, are described below.

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".

Static Analysis

Once a program has passed through manual testing steps, it can be tested in more depth. 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 to 30 percent of the latent software defects in your program. There are many static analyzers. For instance, static analyzers include tools for detecting data element misuse and complexity measurement tools, which estimate the difficulty of testing and help identify hard to test modules with a statistic, or finally, conformance measure tools, which flag confusing or inefficient code.

Dynamic Analysis

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

To obtain statistics on the program parts that have been covered by your tests can often be very difficult. Dynamic analysis can uncover 85 percent

to 90 percent of the potential remaining software defects. SR's *TCAT-PATH*, for instance, will produce data on what has been validated and what has been left out of your testing.

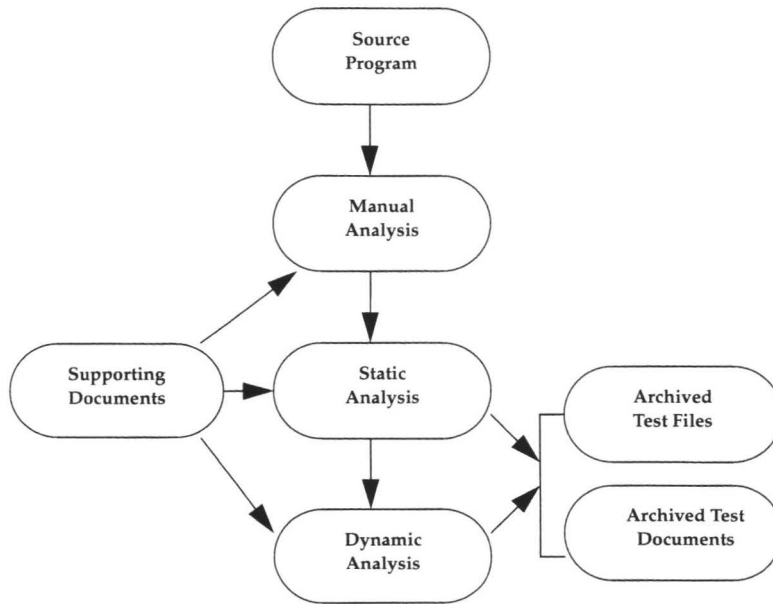


FIGURE 3 Stages in Software Testing

1.6 Single- and Multiple-Module Testing

Another consideration in getting the most out of *TCAT* 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* keeps track of each module by name.

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

Bottom-Up

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.

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* 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* focuses specifically on function calls within the system, so that the tester can verify each interconnection.

1.7 TCAT's Cost Benefits

TCAT will save your organization much time and effort. As a matter of fact, the economics of coverage analysis are extremely favorable. Here are some ways it can save you money.

Improved Error Detection

TCAT 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 percent 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 percent of the errors.

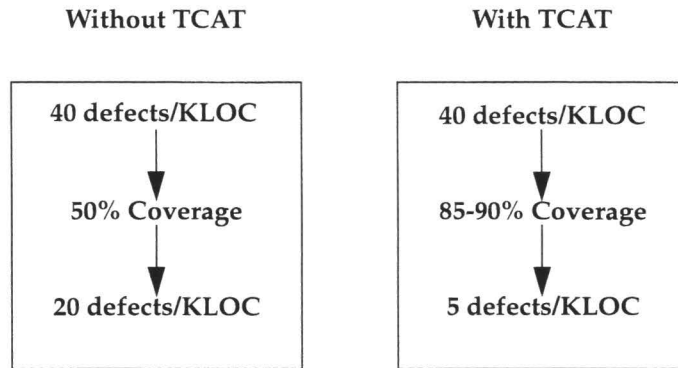


FIGURE 4 Cost Benefit Analysis

The economic value of the increased error detection will vary 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*, 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.

Earlier Error Detection

Not only are more errors detected with *TCAT*, they are also discovered earlier. It's a well accepted truth in Software Development that 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. A classic example of this is the following by Barry Boehm (see Chapter 21, "References"):

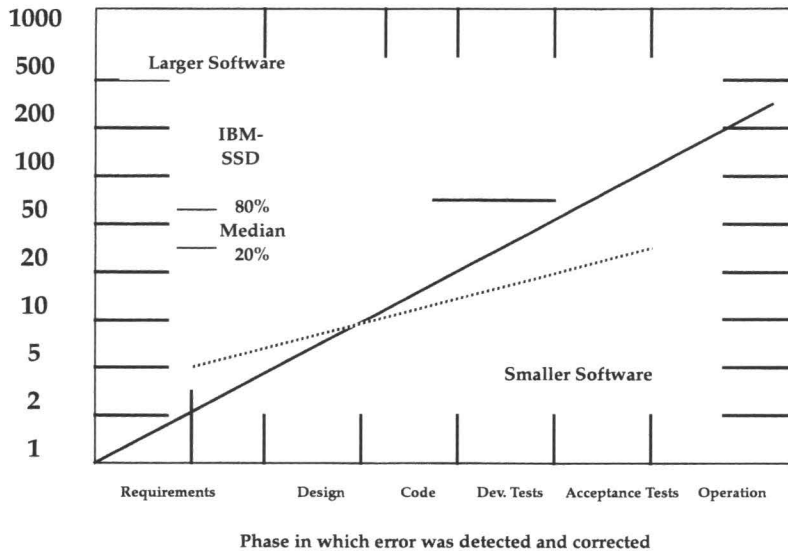


FIGURE 5 Increase in Cost-to-fix Throughout Life-cycle

Your organization can reduce its cost-to-fix ratio by a factor of ten by using *TCAT* 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 much 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 cost.

More Efficient Testing

Using *TCAT*, you can gain in guiding test case development. In general, the tool may be used to identify features that have been missed by existing test suites. The missing items can direct the addition of new test cases.

For example: a software test engineer from a super-minicomputer manufacturer used *TCAT* to reduce the time to test by a factor of eight. As detailed in a technical article available from SR ("References" # 2- 3), he was in charge of testing a C compiler and used *TCAT* to identify the features missed by commercially-available test suites. He 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.

Minimal Test Set

TCAT 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 will exercise all the logic of a system, since test sets require much time and resource to run.

We recommend the use of *SMARTS*, *CAPBAK*, and *EXDIFF* to automate test suite execution, evaluation and analysis steps. These tools can significantly reduce the cost of test suite execution and analysis. *TCAT* 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 under a certain specified coverage threshold, say five percent, for example, it might be reasonable to discard it. Having done so, the tester will end up with a better and easier-to-run test suite.

Assessment of Progress

Coverage analysis with *TCAT* 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 may weigh are the added test cost and the risk of defects.

Coverage analysis data is 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 product latent defect rates. We encourage SQA managers to keep careful records of errors found and corresponding coverage values.



Quick Start

This tutorial chapter gives a quick demonstration of *TCAT* functions.

2.1 Recommendations

It is recommended that you complete the instructions in this chapter before continuing to other sections. This will give you a feel for how the system is organized and will permit you to perform coverage analysis testing.

For best results, follow the instructions very carefully. When you have completed this chapter, you should be familiar with the main activities involved in using *TCAT*, including instrumenting, compiling, linking and running the target program, and finally, looking at resulting coverage reports.

If you are a first-time *TCAT* user, this chapter is best used if you make reference to Chapter 3 for an overview of what is happening at each stage and to Chapter 4 for in-depth operational instructions. If you are an intermediate user, this chapter is best used if you make reference only to those menu definitions which need further explanation (see Chapter 4 and Chapter 5 for further information).

2.1.1 STEP 1: Starting Up TCAT

Before you begin, make sure you are in the X Window System running a window manager (e.g. mwm, olwm, etc.) You should start with the screen organized in a particular way, as shown in Figure 6.

Initialize an xterm-type window by using the mouse to click on New Windows or issuing the command `xterm &` from an existing window. The xterm window will serve as the *TCAT* invocation window.

Move the window to the upper left of the screen. Go to the `$SR/demos/coverage/C/tcat.C` directory. The demos directory is supplied with the product and it consists of an example C program, *example.c*.

This application allows you to select from several types of foods. By selecting various foods, you are actually exercising various logical branches (or segments) of the example program. The goal is to achieve the highest amount of C1 (logical branch) coverage possible for this program through your input. The more selections you make, the higher the coverage.

When initiating this quick start session, your display should look like this:

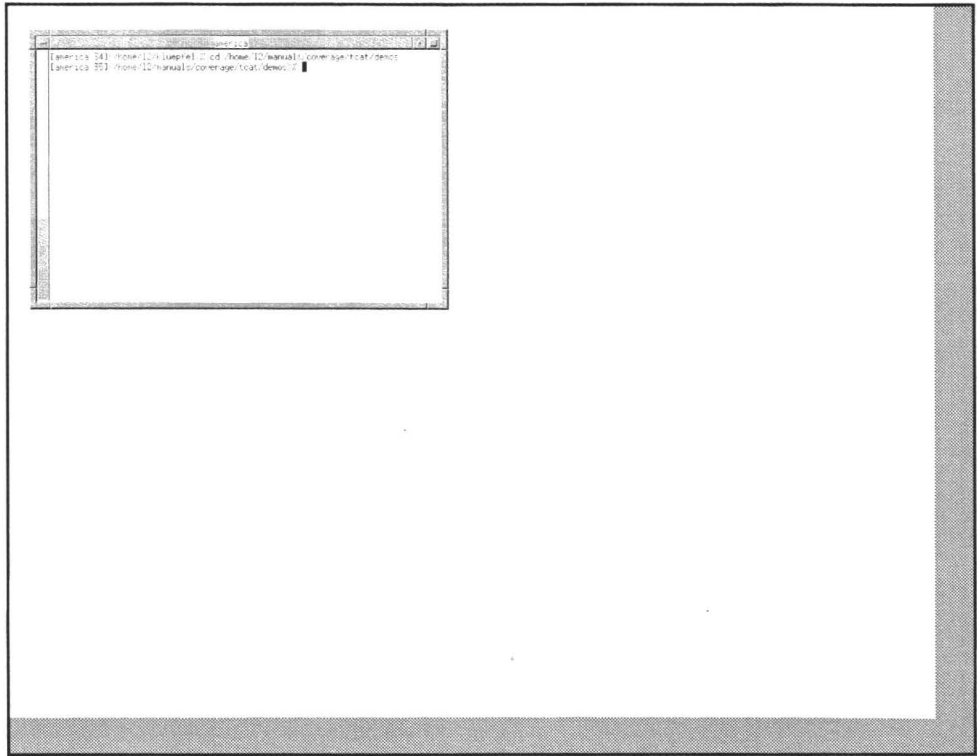


FIGURE 6 Setting Up the Display (Initial Condition)

2.1.2 STEP 2: Invoking TCAT

Now, invoke *TCAT*.

1. Position the mouse pointer, so that it is located in the invocation window.
2. Activate it by clicking the mouse pointer on it. This window becomes the main control window. During your session, all status messages and warnings are displayed in this window.
3. To invoke *TCAT*, type in
Xtcat
Xtcat is the GUI-version of *TCAT*. See Chapter 8 for command line instructions.
4. When you type in this command, the *TCAT* invocation window pops up.
5. Move the *TCAT* invocation window to the upper right of the screen. You can move a window by clicking on its title bar and dragging it.
6. If you want to start over, you can terminate *TCAT* from the *TCAT* invocation window, by clicking on the **System** pull-down menu and selecting **Exit**.

When invoking TCAT, your display should look like this:

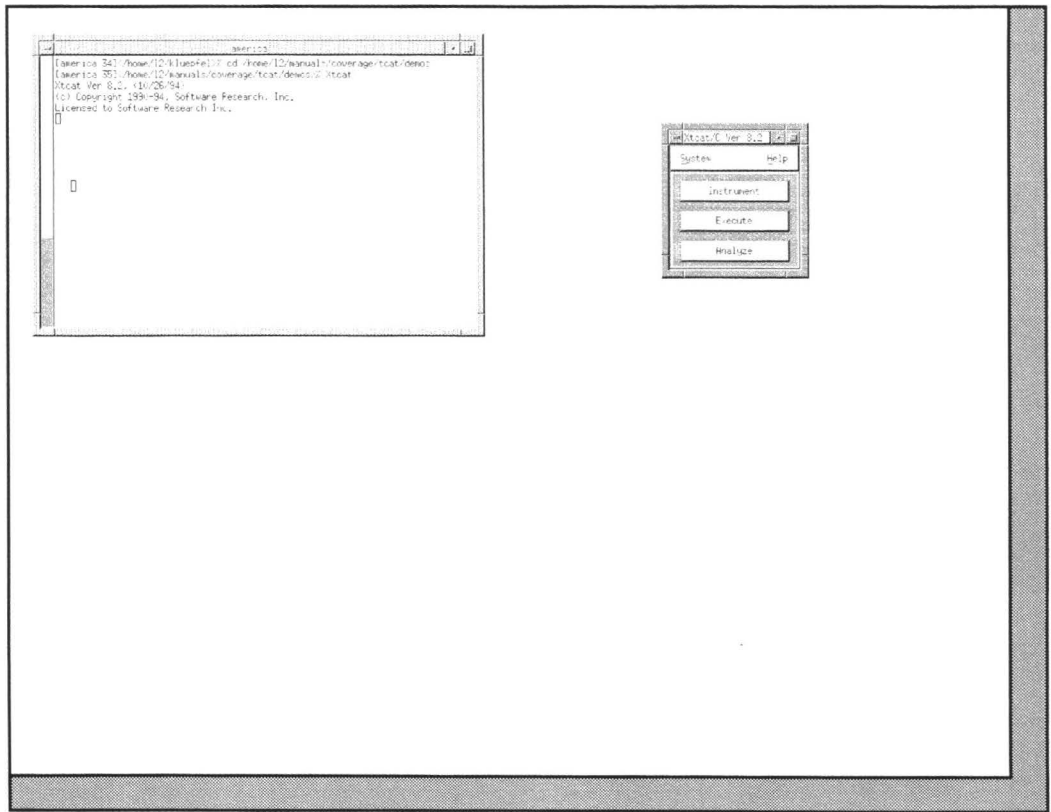


FIGURE 7 Invoking TCAT

2.1.3 **STEP 3: Opening the Instrument Window**

With the **Instrument** window, *TCAT* can automatically instrument the *example.c* program. *TCAT* modifies the source program so that special markers are positioned at every segment in each program module. To invoke:

1. Click on the *TCAT* invocation window's **Instrument** button.
2. The **Instrument** window pops up.
3. Use the mouse to drag the window below the *TCAT* invocation window.

After initializing the **Instrument** window, your display should look like this:

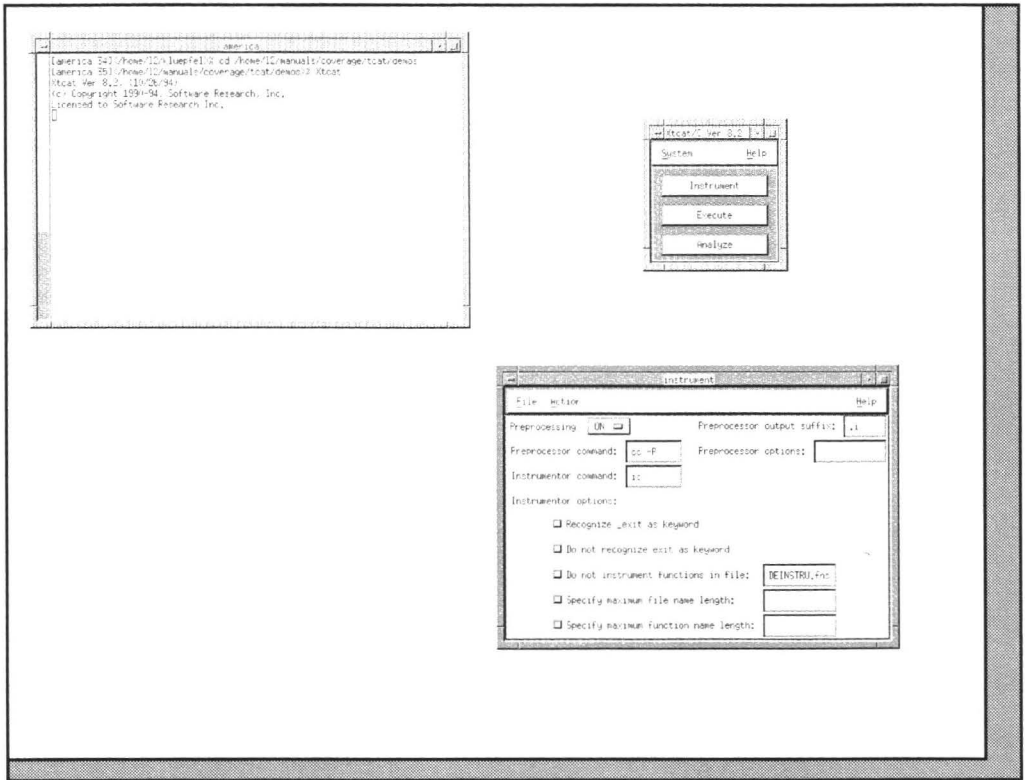


FIGURE 8 Initializing the Instrument Window

2.1.4 STEP 4: Choosing a Target Program

To instrument the supplied example program, you must first select the source file name:

1. Click on the **File** pull-down menu.
2. Select the **Set File Name** option.
3. A file selection dialog box pops up.
4. To select the file, do one of three things:
 - Double click on *example.c* in the **File** selection window.
 - Highlight *example.c* in the **File** selection window or type in the file name in the **Selection** entry box and click on **OK**, or
 - Highlight or type in *example.c* and press the <ENTER> key.

When selecting a target program, your display should look like the one below:

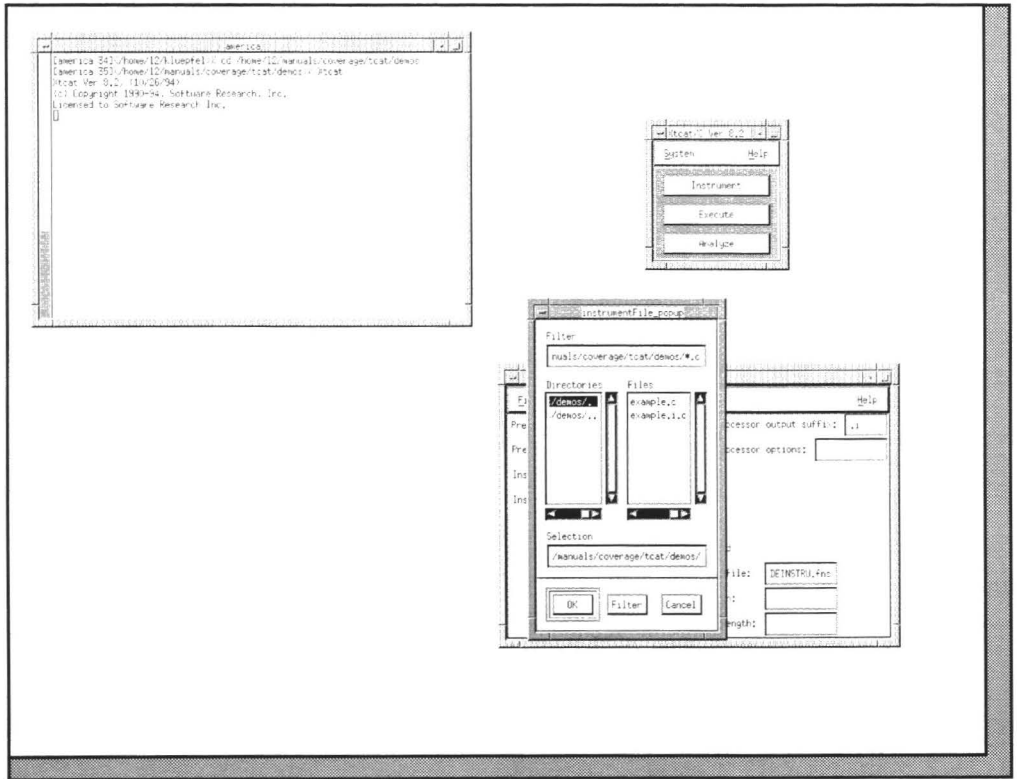


FIGURE 9 Selecting a Target Program

2.1.5 STEP 5: Running the Preprocessor

Before passing the application to the instrumentor, you must first preprocess it.

1. Click on the **Action** pull-down menu.
2. Select **Preprocess**.
3. As *TCAT* preprocesses the source file, *TCAT*'s windows will appear stippled, the mouse pointer changes into a wristwatch symbol and the options gray out. This signifies a time-out period, in which you are unable to select any options until *TCAT* finishes preprocessing.
4. Preprocessing creates an output file named *example.i*.
5. When the mouse pointer symbol returns, preprocessing is complete.

NOTE: If the **Preprocessing** button in the upper-left corner of the **Instrument** window is already toggled to **On**, you may skip this step and proceed to Step 6.

When preprocessing your program, your display should look like this:

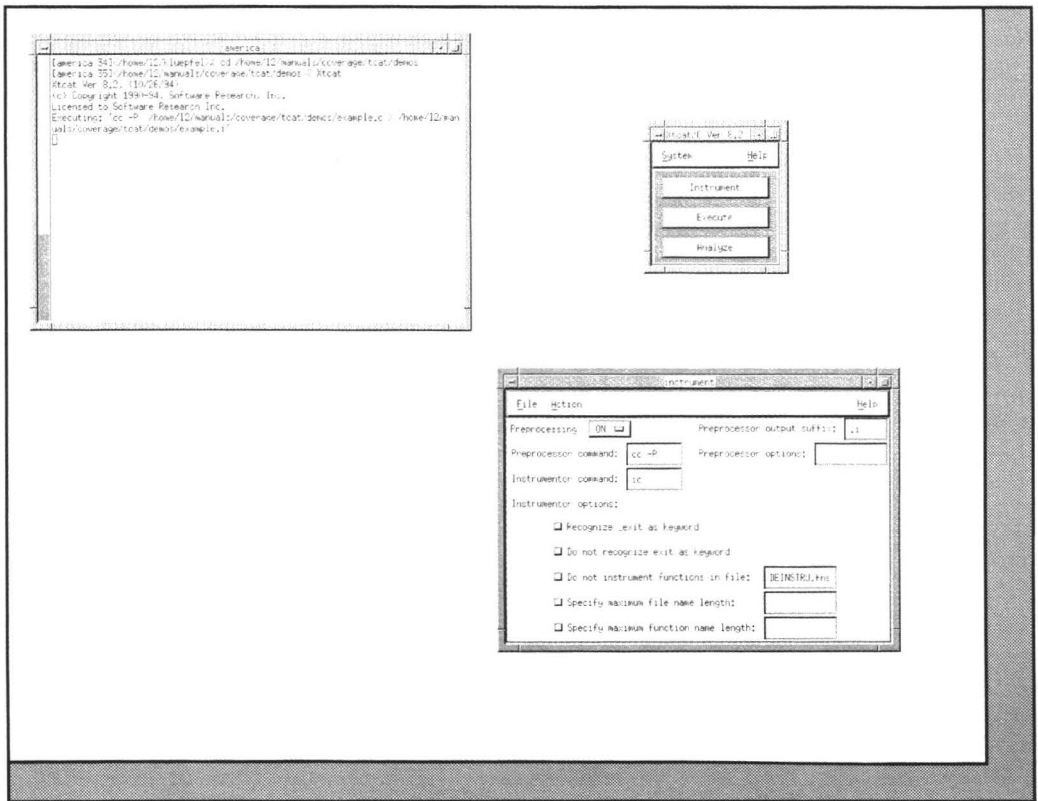


FIGURE 10 Preprocessing Your Program

2.1.6 STEP 6: Instrumenting the Application

After preprocessing, the application is ready for instrumentation. Instrumentation parses the candidate source code, looking for logical branches, or segments. When one is discovered, the instrumentor inserts a function call (a special marker) in the instrumented version of the source code.

Instrumentation produces the following files:

- *basename.i.c* an instrumented version of your "C" program, base-name.
- *basename.i.A*--a **Reference Listing**.
- *basename.i.S*--an **Instrumented Statistics** file.
- *basename.i.L*--a **Segment Count Listing** file.
- *modulename.dig*--a **Directed Graph Listing** file. Each module should have its own *dig* file.
- *basename.i.E*--an **Error Listing** file.

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

1. Click on the **Action** pull-down menu.
2. Select **Instrument**.
3. Like the preprocessing stage, the mouse pointer changes into a wrist-watch symbol and the options gray out.
4. When the mouse pointer returns, the following message should appear in the invocation window:
---> TCAT analysis of 'example' complete, no errors. <---
At this point, instrumentation is complete.
5. Close the **Instrument** window by clicking on the **File** pull-down menu and selecting **Exit**.

After instrumenting your program, your display should look like this:

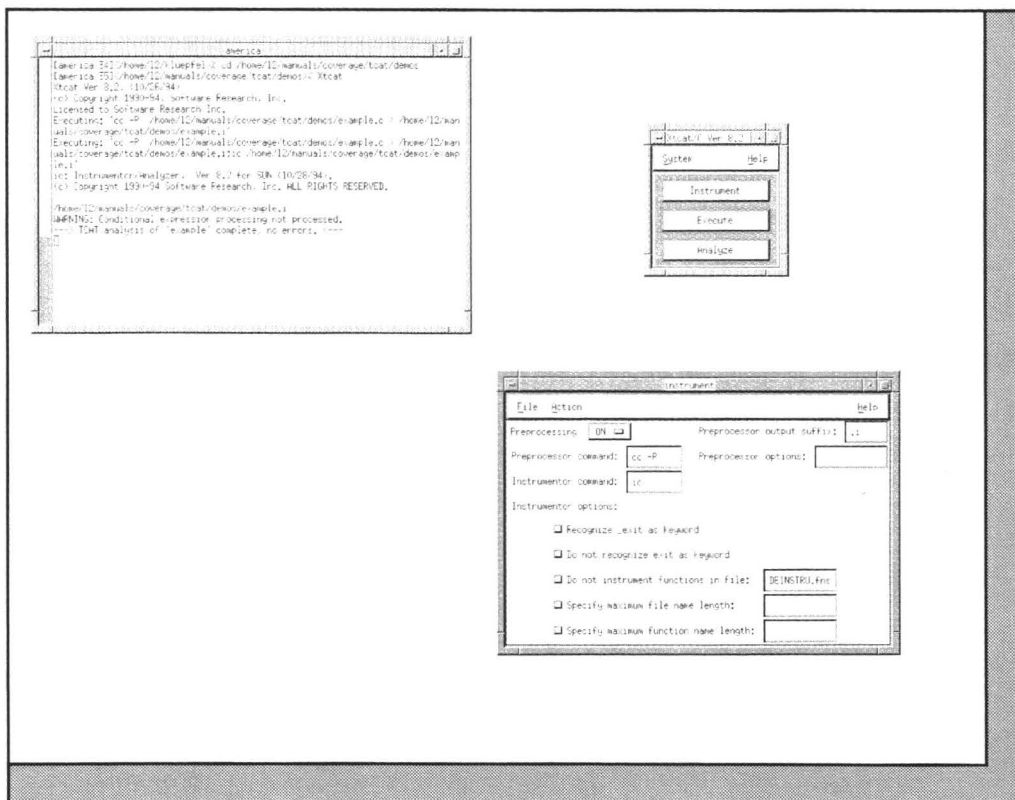


FIGURE 11 Instrumenting Your Program

2.1.7 **STEP 7: Opening the Execute Window**

Once instrumentation is done, you need to compile the instrumented version of your program, link the program's object code to *TCAT/C*'s object modules, or runtime routines, and run the program. This can all be done with the **Execute** window.

To invoke the **Execute** window:

1. Click on the *TCAT* invocation window's **Execute** button.
2. The **Execute** window pops up.
3. Use the mouse to drag the window below the *TCAT* invocation window.

After initializing the Execute window, your display should look like this:

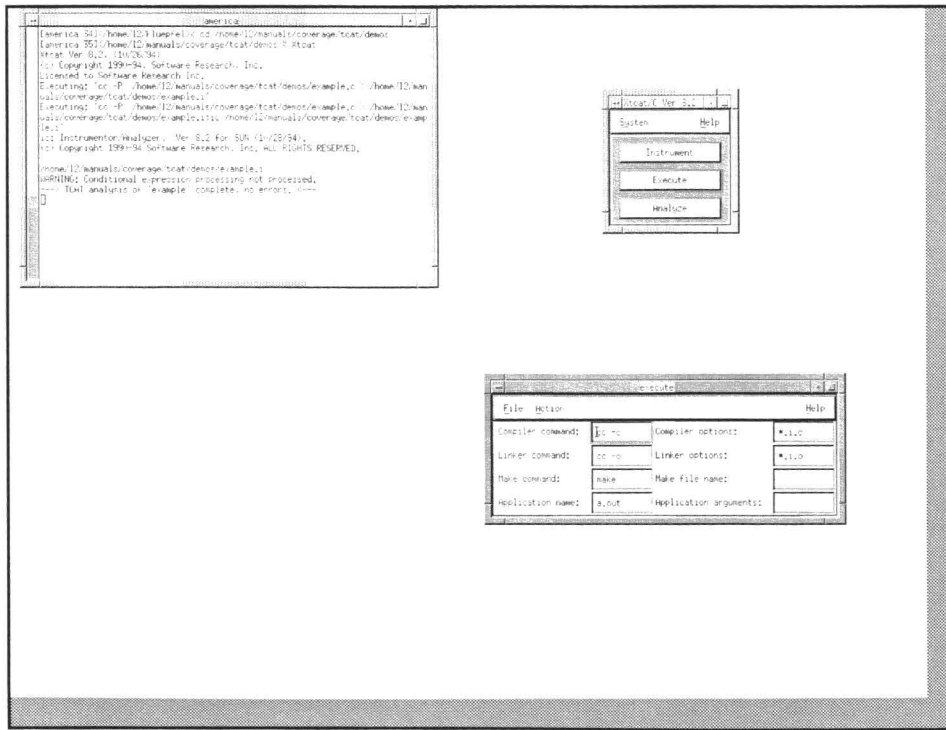


FIGURE 12 Initializing the Execute Window

2.1.8 STEP 8: Compiling

To compile the instrumented version of the example.c program:

1. Click on the **Action** pull-down menu.
2. Select **Compile**.
3. The mouse pointer changes into a wristwatch symbol and the options gray out until instrumentation is complete.

When compiling the instrumented program, your display should look like this:

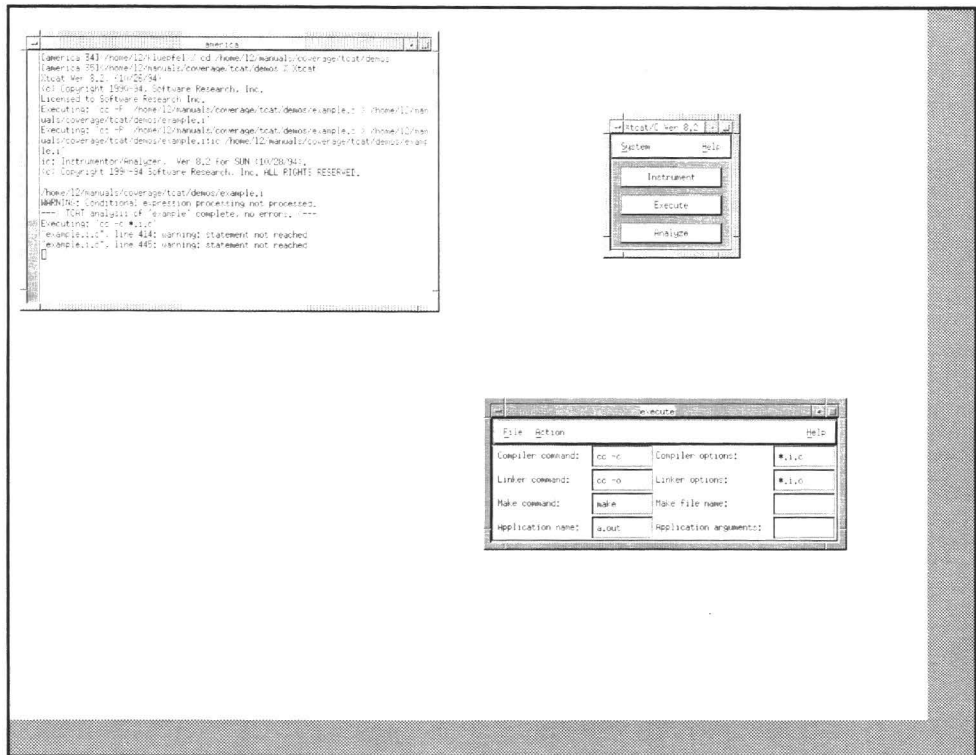


FIGURE 13 Compiling the Instrumented Program

2.1.9 **STEP 9: Choosing a Runtime Version**

In this step, you need to specify the SR-supplied runtime object module you will use to link with your instrumented application's object modules.

SR supplies three runtime object modules:

- *crun0.o* quiet runtime (see the note in STEP 11)
- *crun1.o*
- *cruna.o*

Each runtime object module can change the behavior and the performance of your application. For the purpose of this demonstration, however, use *crun1.o*. To select it:

1. Click on the **File** pull-down menu.
2. Select the **Set Runtime Object Module** option.
3. A file selection dialog box pops up.
4. The three runtime objects modules should be listed in the **Files** selection window.
5. Select *crun1.o* by double-clicking the mouse button on it and then clicking on **OK**. You can also highlight or type in the file name then click on **OK** or press the <ENTER> key.

When selecting a runtime object module, your display should look like this:

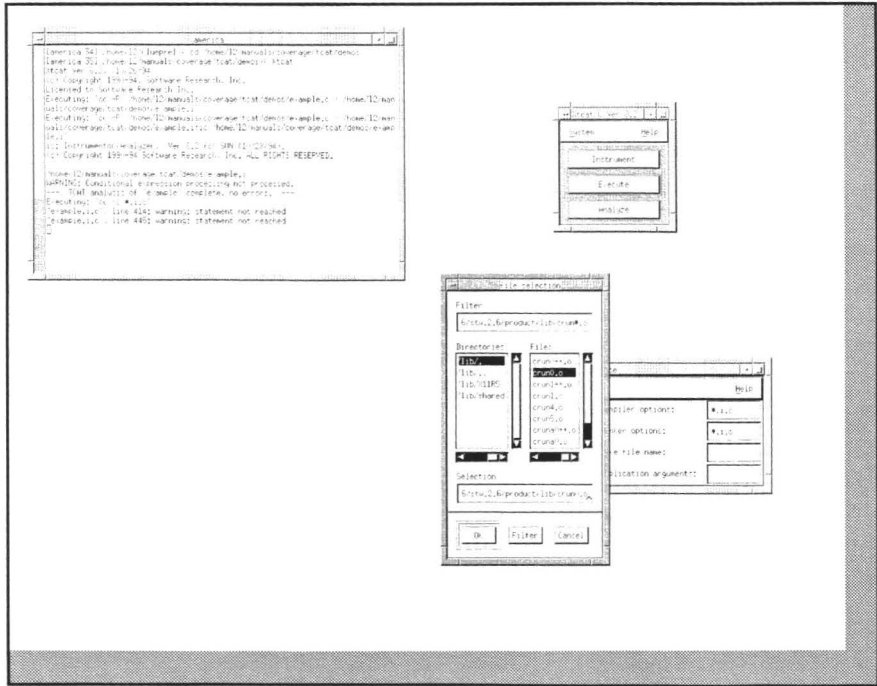


FIGURE 14 Selecting a Runtime Object Module

2.1.10 **STEP 10: Linking the Application**

Now, you are going to link the runtime object module you just selected with the instrumented application's object modules. Linking will create an executable. What you're doing is linking the instructions in the example program to SR's object modules, which records program behavior during execution.

To link:

1. Click on **Action** pull-down menu.
2. Select the **Link** option.
3. The mouse pointer will change into a wristwatch symbol and the options gray out until linking is complete.

When linking the runtime object module to the program's object modules, your display should look like this:

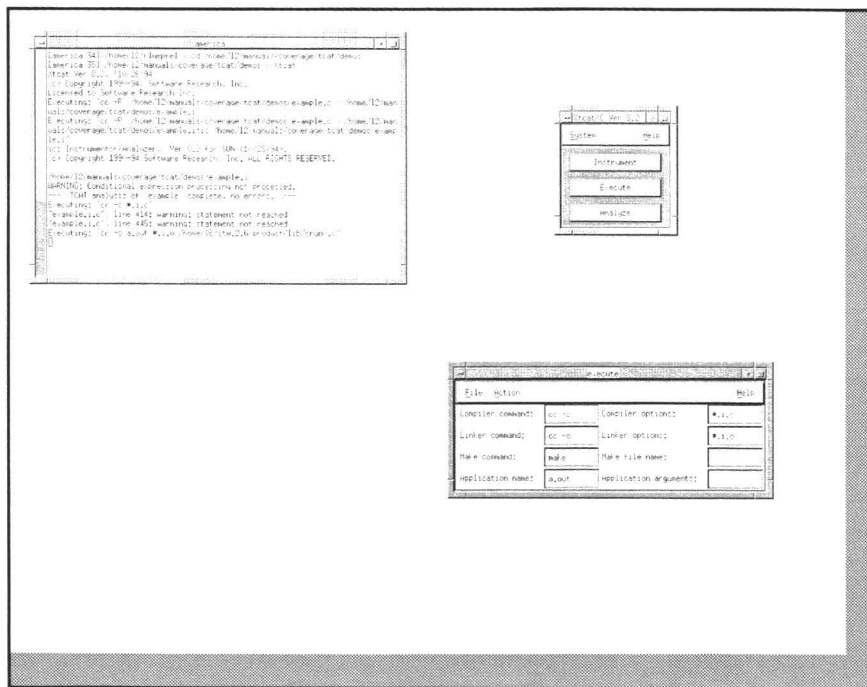


FIGURE 15 Linking Object Modules

2.1.11 STEP 11: Running the Application - Part 1

During instrumentation, *TCAT* inserted function calls at each logical branch it found. In order to later see what the C1 coverage is, you must run the application.

This application is designed to ask you which type of food in the San Francisco, CA area you would like to eat. By selecting particular types of food, you are actually exercising program segments. The more times you run the application and the more types of food you select during each run, the more segments you will hit. This information is then written to a trace file.

To run the application:

1. Click on **Action** pull-down menu.
2. Select the **Run application** option.
3. The mouse pointer will change into a wristwatch symbol and the options will gray out.
4. The application will then prompt you,

Name of tracefile:

[default is *Trace.trc*]

Type in **quick.trc** and then press the <ENTER> key.

5. The invocation window will prompt you,

Trace descriptor:

Activate the window, type in **quick start test**, and then press the <ENTER> key.

Here, the application is asking you to put in a comment about the test. This is particularly useful when you are planning on running several test. For smaller tests, however, it is quickest just to press the <ENTER> key, without typing anything.

Here the application is asking you what trace file name you want your information saved to. Although it is not required, it is important that you set the file to the suffix *.trc*, so you can easily recognize the file as a unique trace file.

NOTE: If you had chosen the quiet runtime *crun0.o*, your test run information would have automatically defaulted to the file *Trace.trc*. You would have not been prompted with the questions in 4 and 5.

When naming the trace file, your display should look like this:

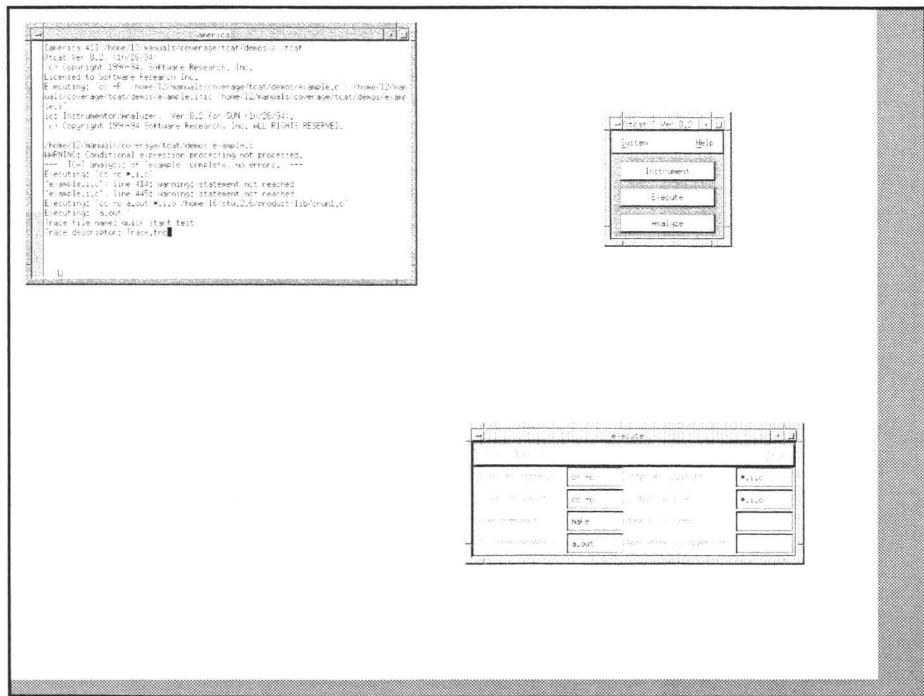


FIGURE 16 Naming the Trace File

2.1.12 STEP 12: Running the Application - Part 2

After specifying the trace file where test run information will be written to, follow these steps:

1. After specifying the trace file name and pressing the <ENTER> key, the example.c program should appear in the invocation window. It asks you,

```
"What type of food would you like?"
```
2. In order to get the most coverage from this run, type in

```
1 2 3 4 5 6 7 8
```

for the eight types of food listed.
3. Press <ENTER>.
4. Eight restaurants that reflect the eight types of food you selected will be displayed.
5. The following message will prompt you,

```
"Do you want to run it again?"
```

During an ordinary testing situations, you would normally run the application a couple of times, selecting various combinations of food types. For now, however, just type in n for no. You'll soon have plenty of opportunities to execute several runs of your own application!
6. The wristwatch symbol will change to the familiar pointer symbol.
7. Close the Execute window by clicking on the **File** pull-down menu and selecting **Exit**.

When running the application, your display should look like this:

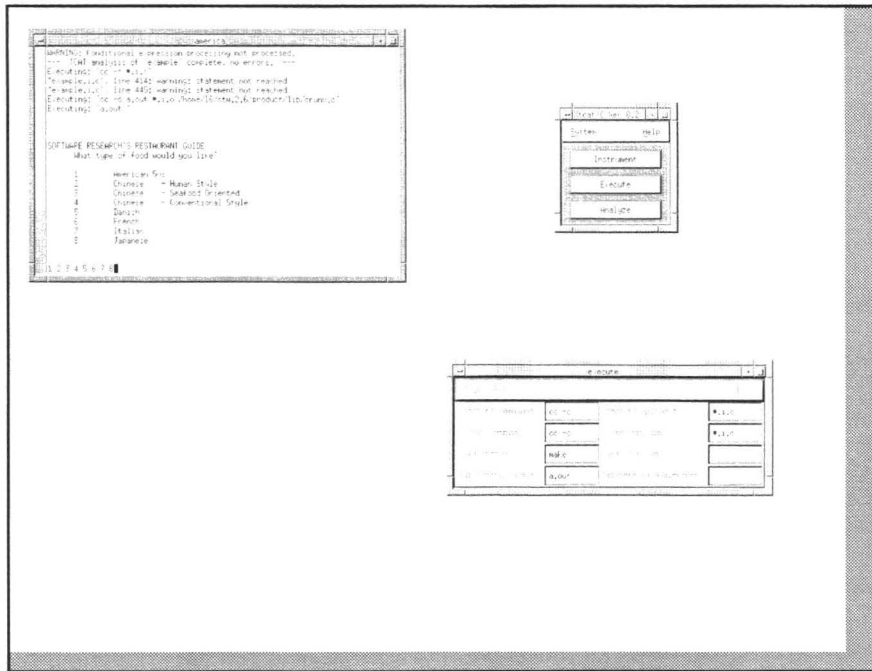


FIGURE 17 Running the Application

2.1.13 STEP 13: Opening the Analyze Window

All the information from the run of the application is stored in the trace file. From the trace file, coverage reports are produced. The **Analyze** window allows you look at several reports, which tell you which segments have or have not been hit.

Here's how to open the **Analyze** window:

1. Click on *TCAT* invocation window's **Analyze** button.
2. The **Analyze** window pops up.
3. Use the mouse to drag the window below the *TCAT* invocation window.

After initializing the **Analyze** window, your display should look like this:

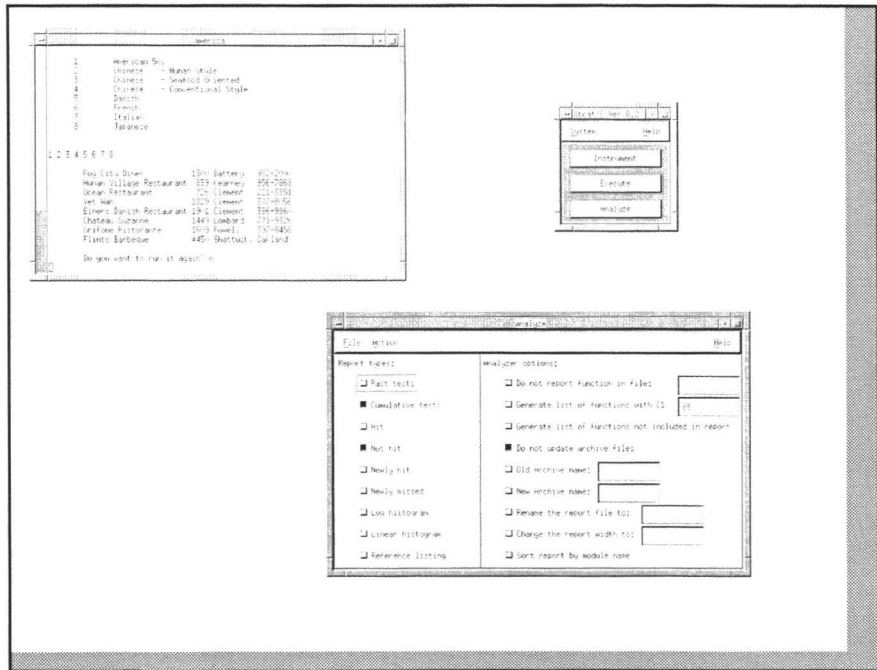


FIGURE 18 Initializing the Analyze Window

2.1.14 STEP 14: Choosing a Trace File

Before looking at coverage reports, you must first select the trace file you specified when running the application, *quick.trc*. Here's how:

1. Click on the **File** pull-down menu.
2. Select **Set Input Trace File Name**.
3. A file selection dialog box pops up.
4. The *quick.trc* file should be listed in the **Files** selection window.
5. Select it by double clicking the mouse button on it and then clicking on **OK**. You can also highlight or type in the file name then click on **OK** or press the <ENTER> key.

When selecting a trace file name, your display should look like this:

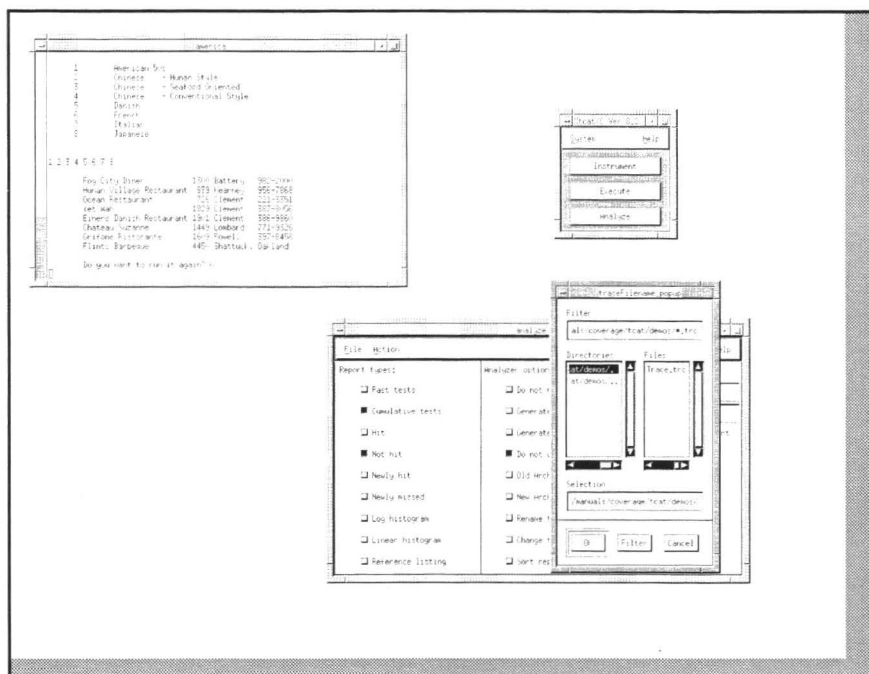


FIGURE 19 Selecting a Trace File Name

2.1.15 STEP 15: Choosing Coverage Reports

From the **Analyze** window, you can look at several different kinds of coverage reports. In general, the **Cumulative**, **Not Hit**, and **Reference Listing** reports are the most frequently looked at coverage reports.

The **Cumulative** report lists each module by name and indicates the number of segments. It tells you how many times each module was invoked, how many times its segments were hit, and its resulting C1 coverage.

The **Not Hit** report shows which segments were not hit. It gives you the module name and identification number for each segment not hit in the current test.

To identify the actual code not executed and plan new test cases, you can look up the segment in the **Reference Listing** report.

1. To select these three reports, simply click on the accompanying check boxes.
2. In the case of the **Reference Listing** report, a file selection dialog box pops up when you click on the check button.
3. A file named *example.i.X* should be listed in the **Files** selection window. This file was created during instrumentation. (Please see STEP 6 for a full explanation.)
4. Select it by double clicking the mouse button on it and then clicking on **OK**. You can also highlight or type in the file name, then click on **OK** or press the <ENTER> key.

When selecting the reference listing file, your display should look like this:

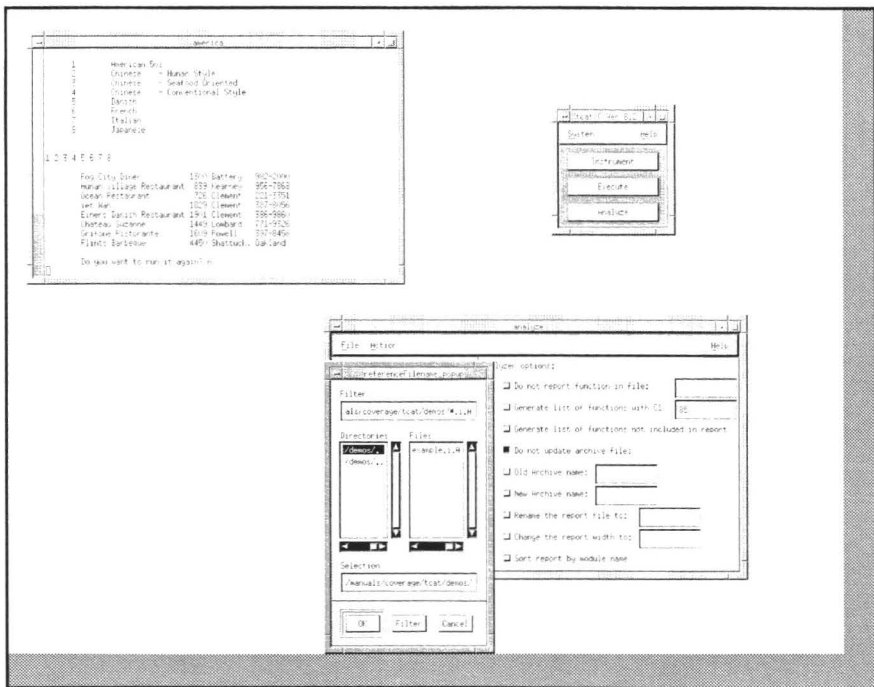


FIGURE 20 Selecting the Reference Listing File

2.1.16 **STEP 16: Viewing the Coverage Reports**

To look at the **Cumulative**, **Not Hit**, and **Reference Listing** reports:

1. Click on the **Action** pull-down menu.
2. Select **Run Coverage Analyzer**.
3. The mouse pointer changes into a wristwatch symbol and the options gray while *TCAT* reads in the trace file and the reference listing to create a report format you can understand.

During this period, the following message appears in the invocation window:

```
Processing date from trace file: quick.trc]...
```

When the information is read in, the mouse pointer returns.

4. Click on the **Action** pull-down menu and select **View Report**.
5. A **Report** window pops up. It first lists a selection status of all the reports. For this demonstration, only three (the ones selected) of the nine possible reports were selected. After this status listing, it contains the **Cumulative** report, the **Not Hit** report, and the **Reference Listing**.
6. Move the window below the invocation window.
7. Use the scroll bars to move side/side and up/down.
8. When finished studying the reports, click on the **Action** pull-down menu and select **Exit**.

When looking at coverage reports, your display should look like this:

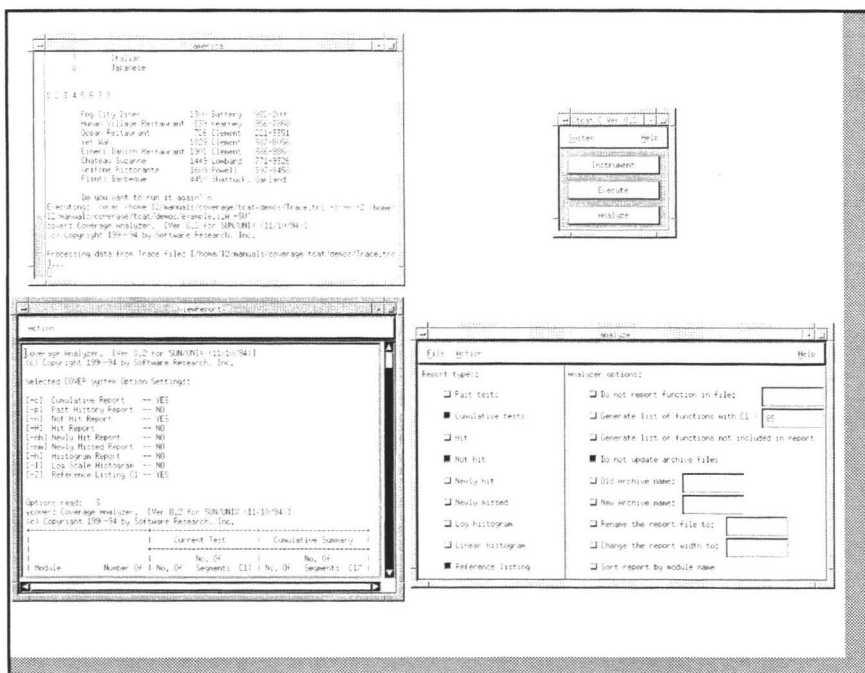


FIGURE 21 Looking at Coverage Reports

2.1.17 **STEP 17: Selecting a Digraph of a Module**

Besides looking at coverage reports after using the **Run Coverage Analyzer** option, you can also look at the source code for a particular segment. To do this, you are first going to select one of the program's modules and then look at the source code for one of the selected module's segments (see STEP 18 for this part).

Here's how:

1. Click on the **Action** pull-down menu.
2. Select **View Source**.
3. A file selection dialog box pops up.
4. The three modules for the application are listed: *chk_char.dig*, *main.dig* and *proc_input.dig*. Each of these modules consists of several segments. For this demonstration, select module *main.dig*.
5. Select it by double clicking the mouse button on it and then clicking on **OK**. You can also highlight or type in the file name and then click on **OK** or press the <ENTER> key.

When selecting a module to graphically view, your display should look like this:

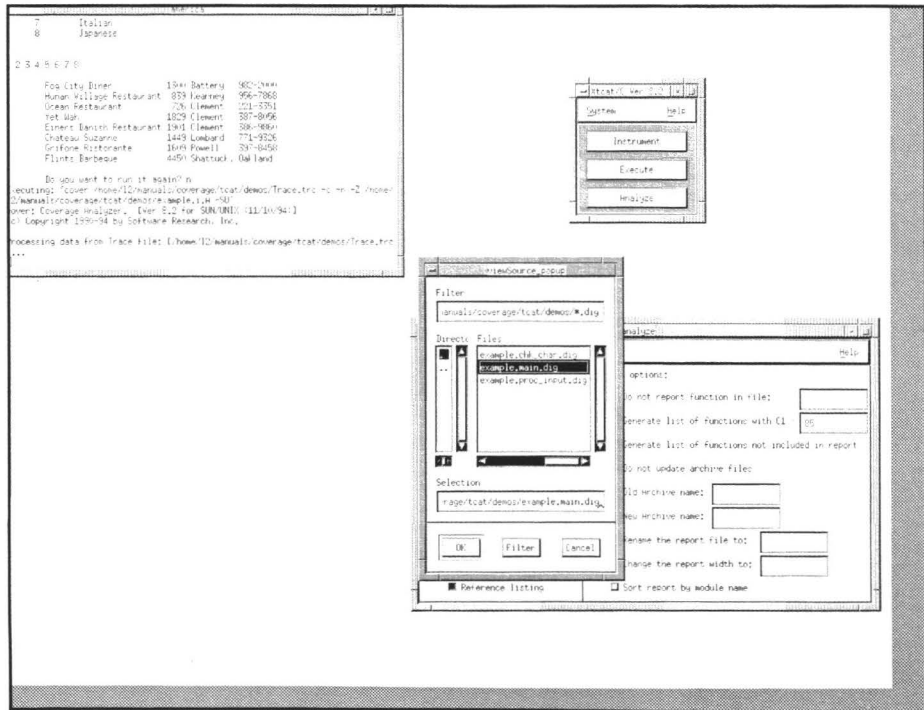


FIGURE 22 Selecting a Module

2.1.18 STEP 18: Viewing a Logical Branch's Source Code

In this step you are going to look at the source code for a particular segment of the module you selected in STEP 17.

1. After selecting the module, a window pops up visually displaying the module. This display is called a directed graph. Its circles represent nodes, or true/false decision points, and the curved lines represent segments.
2. Move the window to the lower left of the screen.
3. Click on the **View Source** pull-down menu.
4. A **View Source** window pops up, which contains the source code for the module.
5. Move the **View Source** window over the **Analyze** window.
6. For this demonstration, you are going to look at the source code for Segment 17. To do so, position the mouse pointer on Segment 17 and press the mouse button.
7. *TCAT* automatically locates the source code for Segment 17 and displays it in the **View Source** window.
8. Feel free to use scroll bars to move up/down or side/side.
9. When you are finished looking at the source code, click on **View Source's Action** pull-down menu and select **Exit**. The window closes.
10. To exit the digraph, click on **File** pull-down menu and select **Exit**. The window closes.

When looking at source code, your display should look like this:

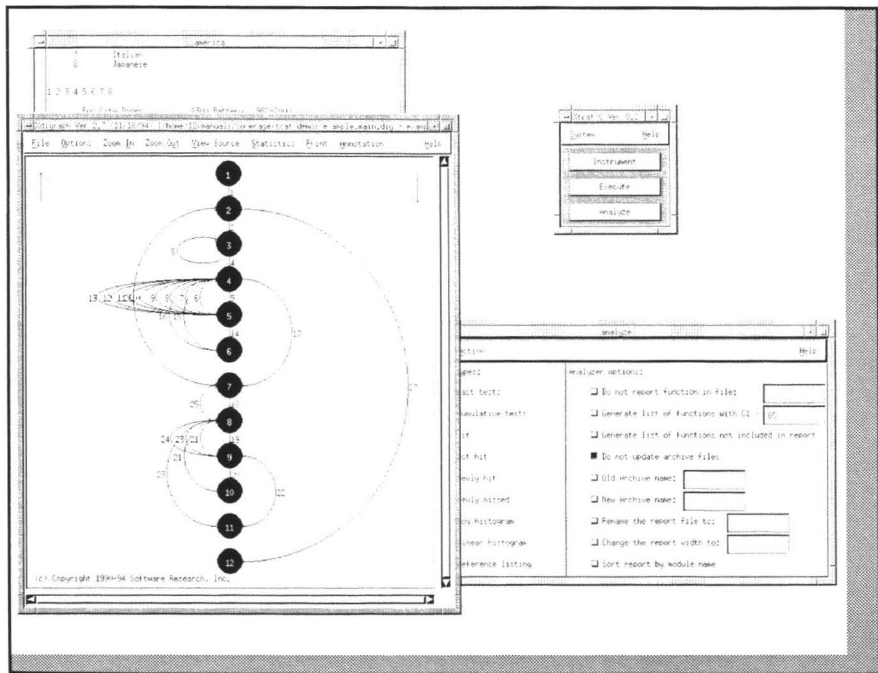


FIGURE 23 Looking at Source Code

2.1.19 STEP 19: Sign Off and Cleanup

After looking at the source code, follow these steps to complete the session:

1. Close the **View Source** file selection pop-up window by clicking on the **Cancel** button.
2. Close the **Analyze** window by clicking on the **File** pull-down menu and selecting **Exit**.
3. Close the **TCAT** invocation window by clicking on the **System** pull-down menu and selecting **Exit**.

When completing your test session, your display should look like this:

```

1  Dallas
2  Johnson

1 2 3 4 5 6 7 8
Fog City Drive      4700  Eastern  902-0101
Hazel Village Restaurant  639  Eastern  955-7650
Ocean Restaurant    725  Eastern  214-3490
Jett Way            1829  Eastern  507-5456
Elliott Davis Restaurant  1911  Eastern  817-8600
Cateau Gardens     1440  Eastern  714-9736
Cypress Parkside   1610  Eastern  214-6450
Flint Barbecue     4485  Eastern  214-3490

Do you want to run it again? n
Executing: C:\msd\manual\coverage\catdemo\Tracefile\demo12.mh (2)
C:\msd\manual\coverage\catdemo\demo12.mh (2)
Coverage analyzer, Ver 3.0, for SUN486 (11/10/94)
Copyright 1991-94 by Software Research, Inc.

Processing data from Trace file: C:\msd\manual\coverage\catdemo\Tracefile\demo12.mh
(Sun486 30) Home-10\manual\coverage\catdemo\1

```

FIGURE 24 Completing a TCAT Session

2.2 Summary

If you successfully completed the preceding 19 steps, you've seen and practiced the basic skills you need to use *TCAT* productively. In this chapter you should have learned how to invoke *TCAT*, how to instrumenting, compile and link, and execute a program, and how to look at coverage reports.

For best learning, you may want to:

- Repeat STEPS 1 - 19 without the manual and experiment by running the application several times and looking at the amount of coverage your test input receives.
- Repeat STEPS 1 - 19 with your application.
- Turn to the chapters on system operation reference and GUI reference where you had difficulties. The table of contents and the index can help you locate the topic you want.

System Introduction

This chapter is an overview of the *TCAT* system, which explains the overall operation of *TCAT* and shows how code is affected at each stage.

LEVEL: If you are an advanced *TCAT* user, you may skip this chapter, which is intended for beginning and intermediate users.

3.1 Overview of *TCAT*

TCAT takes your program and automatically instruments it. During instrumentation, *TCAT* inserts function calls (special markers) at every logical branch (or segment) in each program module. Instrumentation also creates a reference listing file, which is a version of your program which has segment marking comments added to it in a manner similar to the code added to the instrumented version. Extensive logical branch notation 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. If the reports indicate that you have less than 85 percent coverage (the recommended amount), you can identify unexercised logical branches by looking at the entire reference listing report or you can look at the reference listing code for a particular logical branch. When you identify the troubled areas, you can then create new test cases and re-execute the program.

TCAT can help you reach your goal: creating the most extensive test cases possible.

3.2 How to Use *TCAT*

To obtain coverage for your program, you should follow these steps:

4. Preprocess.
5. Instrument Program Code (marking logical branches).
6. Compile and Link Code (recording and counting markers).
7. Execute Program and Generate Trace File.

8. Generate Coverage Reports (reporting logical branches hit).

To explain the various stages, we wrote a simple program named `example.c`. This program asks you questions about which type of cuisine in the San Francisco, CA area you would like to eat. If you went through the Chapter 2, you may already have a feel for the program.

This program consists of three function modules: `main`, `proc_input` and `chk_char`. Please take note of the following points:

- Point A Marks the `#include` statement that imports the standard input-output `stdio.h` code. This will also be included in the instrumented C program file as well as in the Reference Listing file.
- Point B Indicates the main function with its `argc` and `argv` arguments.
- Point C Refers to the two function names, `proc_input` and `chk_char`.

```

/* EXAMPLE.C --example file for use with TCAT, STCAT, TCAT-PATH. */
+-----+
|#include "stdio.h" |      A
+-----+
#include <ctype.h>

#define INPUTERROR      -1
#define INPUTDONE       0
#define MENU_CHOICES    13
#define STD_LEN         79
#define TRUE            1
#define FALSE           0
#define BOOL            int
#define OK               TRUE
#define NOT_OK          FALSE

char menu[MENU_CHOICES][STD_LEN] = {
    "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
    "    What type of food would you like?\n",
    "\n",
    "    1      American 50s  \n",
    "    2      Chinese    - Hunan Style \n",
    "    3      Chinese    - Seafood Oriented \n",
    "    4      Chinese    - Conventional Style \n",
    "    5      Danish     \n",
    "    6      French     \n",
    "    7      Italian    \n",
    "    8      Japanese   \n",
    "\n\n"
}

```

```
);
int char_index;
+-----+
|#include <ctype.h> |      B
+-----+
main(argc,argv)      /* simple program to pick a restaurant */
int  argc;
char  *argv[];
{
    int  i, choice, c,answer;
    char str[STD_LEN];
    BOOL ask, repeat;
    int  proc_input();
    c = 3;
    repeat = TRUE;
    while(repeat) {
        printf("\n\n");
        for(i = 0; i < MENU_CHOICES; i++)
            printf("%s", menu[i]);
        gets(str);
        printf("\n");
        while(choice = proc_input(str)) {
            switch(choice) {
                case 1:
                    printf("\tFog City Diner 1300 Battery    982-2000 \n");
                    break;
                case 2:
                    printf("\tHunan Village Restaurant  839 Kearney  956-7868
\n");
                    break;
                case 3:
                    printf("\tOcean Restaurant  726 Clement    221-3351 \n");
                    break;
                case 4:
                    printf("\tYet Wah  1829 Clement    387-8056 \n");
                    break;
                case 5:
                    printf("\tEiners Danish Rest. 1901 Clement 386-9860 \n");
                    break;
                case 6:
                    printf("\tChateau Suzanne  1449 Lombard 771-9326 \n");
                    break;
                case 7:
                    printf("\tGrifone Ristorante 1609 Powell    397-8458
\n");
                    break;
                case 8:
                    printf("\tFlints Barbecue 4450 Shattuck, Oakland \n");
                    break;
            }
        }
    }
}
```

```
        default:
            if(choice != INPUTERROR)
                printf("\t>>> %d: not a valid choice.\n", choice);

    } }
for(ask = TRUE; ask; ) {
    printf("\n\tDo you want to run it again? ");
    while((answer = getchar()) != '\n') {
        switch(answer) {
            case 'Y':
            case 'y':
                ask = FALSE;
                char_index = 0;
                break;
            case 'N':
            case 'n':
                ask = FALSE;
                repeat = FALSE;
                break;
            default:
                break;
        } } } } }

+-----+
|int proc_input(in_str)|      C
+-----+
char *in_str;
{
    int tempresult = 0;
    char bad_str[80], *bad_input;
    BOOL got_first = FALSE;
    bad_input = bad_str;
    while(isspace(in_str[char_index]))
        char_index++;
    for( ; char_index <= strlen(in_str); char_index++) {
        switch(in_str[char_index]) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                /* process choice */
                tempresult = tempresult * 10 + (in_str[char_index] - '0');
                got_first = TRUE;
                break;
        }
    }
}
```

```

default:
    if(chk_char(in_str[char_index])) {
        return(tempresult);
    }
    else {
        if(char_index > 0 && got_first)
            char_index--;
        while(char_index <= strlen(in_str)) {
            if(chk_char(in_str[char_index]))
                break;
            else
                *bad_input++ = in_str[char_index];
            char_index++;
        }
        *bad_input = '\0';
        printf("\t>>> bad input: %s\n", bad_str);
        char_index++;
        return(INPUTERROR);
    } } }
return(INPUTDONE);
}
+-----+
|BOOL chk_char(ch) |      C
+-----+
char ch;
{
    if(isspace(ch) || ch == '\0')
        return(OK);
    else
        return(NOT_OK);
}

```

FIGURE 25 Sample C Program

3.2.1 Preprocessing Source Code

Most often, you must check your program for syntax errors by preprocessing. When you preprocess, your source code file *basename.c* is automatically copied to a file named *basename.i* which is where the preprocessing takes place.

3.2.2 Instrument Program Code

The second step in analyzing test coverage with *TCAT* is to instrument the source code. *TCAT* modifies the program so that special markers are positioned at every logical branch in each program module. Later, during program execution, these markers will be tracked and counted by *TCAT*

to provide data for coverage analysis. Instrumentation does not affect a program's logical behavior, although it increases test execution time and code size by about 20 percent.

During instrumentation, *TCAT* generates several files:

- *example.i.c* -- an Instrumented Version of your C program.
- *example.i.A* -- Reference Listing, showing where
- *TCAT* has placed each logical branch marker and how they are numbered.
- *example.i.S* -- Instrumented Statistics Listing.
- *example.i.L* -- Segment Count Listing.
- *modulename.dig* -- Directed Graph Listing for each module.
- *example.i.E* -- Error Report.

Examples of the above files are shown next.

When you instrument *basename.i*, your instrumented program is automatically copied to a file named *basename.i.c*, which is where the instrumentation markers are placed. In the case of *example.c*, the file name becomes *example.i.c*

The effect of instrumentation on the *example.c* program are displayed in boxes and shown in bold face on the following pages. Please take note of the following points:

- | | |
|---------|---|
| Point A | Marks the specific header information for this copy of <i>TCAT</i> . Included are the system release number and information on the copyright and licensing agreement. |
| Point B | Refers to the runtime modules. |
| Point C | Traces the start of the program. |
| Point D | Traces the start or entry of a function. |
| Point E | Traces a segment. The number identifies the branch number; this number is transmitted to the trace file when the instrumented program is run. |
| Point F | Traces the exit of the function. |
| Point G | Traces the exit of the program. |

```

+-----+
|                                             |
|-----|
|/*                                           |
|-- C1 instrumentation by TCAT instrumenter: |
|--                                           |

```

```

|-- Program ic, Release 8.2
|
|-- SR Copy Identification No. 0.
|--
|-----|
|      A
|-- (c) Copyright 1990 by Software Research, Inc. All Rights Reserved. |
|--
|-- This program was instrumented by SR proprietary software,
|-- for use with the SR proprietary TCAT runtime package.
|-- Use of this program is limited by associated software
|-- license agreement.
|-----|
|*/
|-----+
+-----+
|extern SegHit();      |
|extern Strace();      |
|extern Ftrace();      |      B
|extern EntrMod();     |
|extern ExtMod();      |
+-----+

char menu[13][79] = {
    "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
    "    What type of food would you like?\n",
    "\n",
    "    1      American 50s  \n",
    "    2      Chinese    - Hunan Style \n",
    "    3      Chinese    - Seafood Oriented \n",
    "    4      Chinese    - Conventional Style \n",
    "    5      Danish      \n",
    "    6      French      \n",
    "    7      Italian     \n",
    "    8      Japanese   \n",
    "\n\n"
};

int char_index;

main(argc,argv)
int  argc;
char *argv[];
{
    int i, choice, c,answer;
    char str[79];
    int ask, repeat;
    int proc_input();

+-----+
|   Strace("IC",0x7504,0,0);   |      C
+-----+
+-----+

```

```
|   EntrMod(27,"main",-1); |   D
+-----+
+-----+
|   SegHit(1); |   E
+-----+
c = 3;
repeat = 1;
{ while(repeat) { SegHit(2);
  {
    printf("\n\n"); {
    for(i = 0; i < 13; i++) { SegHit(3);
      printf("%s", menu[i]); }
    SegHit(4); };
    gets(str);
    printf("\n");

    { while(choice = proc_input(str)) { SegHit(5);
      {
        { switch(choice)

          case 1: SegHit(6);
            printf("\tFog City Diner   1300 Battery   982-2000 \n");
            break;
          case 2: SegHit(7);
            printf("\tHunan Village Rest.839 Kearney 956-7868 \n");
            break;
          case 3: SegHit(8);
            printf("\tOcean Restaurant  726 Clement   221-3351 \n");
            break;
          case 4: SegHit(9);
            printf("\tYet Wah 1829 Clement   387-8056 \n");
            break;
          case 5: SegHit(10);
            printf("\tEiners Danish Rest 1901 Clement   386-9860 \n");
            break;
          case 6: SegHit(11);
            printf("\tChateau Suzanne  1449 Lombard 771-9326 \n");
            break;
          case 7: SegHit(12);
            printf("\tGrifone Ristorante 1609 Powell  397-8458 \n");
            break;
          case 8: SegHit(13);
            printf("\tFlints Barbecue 4450 Shattuck, Oakland \n");
            break;
          default: SegHit(14);
            if(choice != -1) { SegHit(15);
              printf("\t>>> %d: not a valid choice.\n", choice);
            } else SegHit(16);
            break;
          } } } } SegHit(17); };

        { for(ask = 1; ask; ) { SegHit(18);
          {
```

```

printf("\n\tDo you want to run it again? ");
{ while((answer = getchar()) != '\n') { SegHit(19);
  {
    { switch(answer)
      {
        case 'Y': SegHit(20);
        case 'y': SegHit(21);
          ask = 0;
          char_index = 0;
          break;
        case 'N': SegHit(22);
        case 'n': SegHit(23);
          ask = 0;
          repeat = 0;
          break;
        default: SegHit(24);
          break;
        } }
      } } SegHit(25); };
    } } SegHit(26); };
  } } SegHit(27); };
+-----+
| ExtMod("main");          | F
+-----+

+-----+
| Ftrace(0);              | G
+-----+
}

int proc_input(in_str)
char *in_str;
{
  int tempresult = 0;
  char bad_str[80], *bad_input;
  int got_first = 0;

  EntrMod(24, "proc_input", -1);
  SegHit(1);

  bad_input = bad_str;

  { while(isspace(in_str[char_index])) { SegHit(2);
    char_index++; } SegHit(3); };

  { for( ; char_index <= strlen(in_str); char_index++) { SegHit(4);
    {
      { switch(in_str[char_index])
        {
          case '0': SegHit(5);
          case '1': SegHit(6);
          case '2': SegHit(7);
          case '3': SegHit(8);

```

```

        case '4': SegHit(9);
        case '5': SegHit(10);
        case '6': SegHit(11);
        case '7': SegHit(12);
        case '8': SegHit(13);
        case '9': SegHit(14);
            tempresult = tempresult * 10 + (in_str[char_index] - '0');
            got_first = 1;
            break;
        default: SegHit(15);
            if(chk_char(in_str[char_index])) { SegHit(16);
                { ExtMod("proc_input");
                    return(tempresult); }
                }
            else { SegHit(17);
                {
                    if(char_index > 0 && got_first) { SegHit(18);
                        char_index--; } else SegHit(19);
                        { while(char_index <= strlen(in_str)) { SegHit(20);
                            {
                                if(chk_char(in_str[char_index])) { SegHit(21);
                                    break; }
                                else { SegHit(22);
                                    *bad_input++ = in_str[char_index]; }
                                    char_index++;
                                } } SegHit(23); }
                                *bad_input = '\0';
                                printf("\t>>> bad input: %s\n", bad_str);
                                char_index++;
                                { ExtMod("proc_input");
                                    return(-1); }
                                } }
                    } }
                } } SegHit(24); };

        { ExtMod("proc_input");
            return(0); }

        ExtMod("proc_input");
    }

int chk_char(ch)
char ch;
{
    EntrMod(3, "chk_char", -1);
    SegHit(1);

    if(isspace(ch) || ch == '\0') { SegHit(2); { ExtMod("chk_char");
        return(1); } }
    else { SegHit(3); { ExtMod("chk_char");
        return(0); } }
}

```

```

    ExtMod("chk_char");
}

```

FIGURE 26 Instrumented Program

The **Reference Listing** file is a version of C program which marks program segments corresponding to logical branch outcomes. This file is useful when you later look at a **Not Hit** report to see which logical branches were not hit. You can then cross-reference with the **Reference Listing** report. This report has the same information as the **Reference Listing** file, except it identifies the coverage for each module, the number of times each logical branch was hit and which branches were not hit.

During instrumentation the **Reference Listing** file is named *basename.i.A*. For the *example.c* program the file name becomes *example.i.A*. The effect of instrumentation on the *example.i.A* file are displayed in boxes and shown in bold face on the following pages. Please take note of the following points:

- | | |
|---------|--|
| Point A | Marks the title and header information. |
| Point B | Shows where function <i>main</i> execution begins. |
| Point C | Shows where function <i>proc_input</i> begins. |
| Point D | Shows where function <i>chk_char</i> begins. |
| Point E | Indicates the number and/or statement type of each logical branch. |

```

+-----+
|-----|
|-----|
|-- TCAT/C, Release 8.2
|
|--
|-- (c) Copyright 1990 by Software Research, Inc. ALL RIGHTS
RESERVED. |
|--
|-- SEGMENT REFERENCE LISTING
|
|--
|-- Instrumentation date: Tue May 4 15:06:02
1993          |      A
|--
|-- Separate modules and segment definitions for each module
are |
|-- indicated in this commented version of the supplied source file.
|
|-----+
+-----+

char menu[13][79] = {

```

```

"SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
"    What type of food would you like?\n",
"\n",
"    1      American 50s   \n",
"    2      Chinese      - Hunan Style \n",
"    3      Chinese      - Seafood Oriented \n",
"    4      Chinese      - Conventional Style \n",
"    5      Danish        \n",
"    6      French        \n",
"    7      Italian       \n",
"    8      Japanese     \n",
"\n\n"
};
int char_index;
main(argc,argv)
intargc;
char*argv[];
{
    int i, choice, c,answer;
    char str[79];
    int ask, repeat;

+-----+
|/** Module main **/ |   B
+-----+

    int proc_input();
+-----+
|/** Segment 1 <> **/ |   E
+-----+
    c = 3;
    repeat = 1;
    while(repeat) {
/** Segment 2 <start while> **/
        printf("\n\n\n");
        for(i = 0; i < 13; i++)
/** Segment 3 <start for> **/
            printf("%s", menu[i]);
/** Segment 4 <end for> **/
        gets(str);
        printf("\n");
        while(choice = proc_input(str)) {
/** Segment 5 <start while> **/
            switch(choice) {
                case 1:
/** Segment 6 <case alt> **/
                    printf("\tFog City Diner1300 Battery    982-2000 \n");
                    break;
                case 2:

```

```
/** Segment 7 <case alt> **/  
    printf("\tHunan Village Restaurant 839 Kearney 956-  
7868 \n");  
        break;  
    case 3:  
/** Segment 8 <case alt> **/  
    printf("\tOcean Restaurant 726 Clement 221-3351  
\n");  
        break;  
    case 4:  
/** Segment 9 <case alt> **/  
    printf("\tYet Wah 1829 Clement 387-8056 \n");  
        break;  
  
    case 5:  
/** Segment 10 <case alt> **/  
    printf("\tEiners Rest 1901 Clement 386-9860 \n");  
        break;  
    case 6:  
/** Segment 11 <case alt> **/  
    printf("\tChateau Suzanne 1449 Lombard 771-9326 \n");  
        break;  
    case 7:  
/** Segment 12 <case alt> **/  
    printf("\tGrifone Ristorante1609 Powell 397-8458 \n");  
        break;  
    case 8:  
/** Segment 13 <case alt> **/  
    printf("\tFlints Barbecue 4450 Shattuck, Oakland \n");  
        break;  
    default:  
/** Segment 14 <case alt> **/  
        if(choice != -1)  
/** Segment 15 <if> **/  
            printf("\t>>> %d: not a valid choice.\n", choice);  
/** Segment 16 <implied else> **/  
            break;  
        }  
    }  
/** Segment 17 <end while> **/  
    for(ask = 1; ask; ) {  
/** Segment 18 <start for> **/  
        printf("\n\tDo you want to run it again? ");  
        while((answer = getchar()) != '\n') {  
/** Segment 19 <start while> **/  
            switch(answer) {  
                case 'Y':  
/** Segment 20 <case alt> **/  
                    case 'y':
```



```

    /** Segment 21 <case alt> */
        ask = 0;
        char_index = 0;
        break;
        case 'N':
/** Segment 22 <case alt> */
    case 'n':
/** Segment 23 <case alt> */
        ask = 0;
        repeat = 0;
        break;
        default:
/** Segment 24 <case alt> */
        break;
    } } } } }
/** Segment 25 <end while> */
/** Segment 26 <end for> */
/** Segment 27 <end while> */
int proc_input(in_str)
char *in_str;
{
    int tempresult = 0;
    char bad_str[80], *bad_input;
+-----+
|/** Module proc_input */|    C
+-----+
    int got_first = 0;
+-----+
|/** Segment 1 <> */|    E
+-----+
    bad_input = bad_str;
    while(isspace(in_str[char_index]))
/** Segment 2 <start while> */
        char_index++;
/** Segment 3 <end while> */
    for( ; char_index <= strlen(in_str); char_index++) {
/** Segment 4 <start for> */
        switch(in_str[char_index]) {
            case '0':
/** Segment 5 <case alt> */
                case '1':
/** Segment 6 <case alt> */

                case '2':
/** Segment 7 <case alt> */
                case '3':
/** Segment 8 <case alt> */
                case '4':
/** Segment 9 <case alt> */

```

```
        case '5':
/** Segment 10 <case alt> **/
        case '6':
/** Segment 11 <case alt> **/
        case '7':
/** Segment 12 <case alt> **/
        case '8':
/** Segment 13 <case alt> **/
        case '9':
/** Segment 14 <case alt> **/

        tempresult = tempresult * 10 + (in_str[char_index] -
'0');

        got_first = 1;
        break;
        default:
/** Segment 15 <case alt> **/
        if(chk_char(in_str[char_index])) {
/** Segment 16 <if> **/
            return(tempresult);
        }
        else {
/** Segment 17 <else> **/
            if(char_index > 0 && got_first)
/** Segment 18 <if> **/
                char_index--;
/** Segment 19 <implied else> **/
            while(char_index <= strlen(in_str)) {
/** Segment 20 <start while> **/
                if(chk_char(in_str[char_index]))
/** Segment 21 <if> **/
                    break;
                else
/** Segment 22 <else> **/
                    *bad_input++ = in_str[char_index];
                    char_index++;
            }
/** Segment 23 <end while> **/
            *bad_input = '\0';
            printf("\t>>> bad input: %s\n", bad_str);
            char_index++;
            return(-1);
        } } }
/** Segment 24 <end for> **/
    return(0);
}
int chk_char(ch)
char ch;
```

```
+-----+
|/** Module chk_char**/ | D
+-----+

{
+-----+
|/** Segment 1 <> **/ | E
+-----+
    if(isspace(ch) || ch == '\0')
/** Segment 2 <if> **/
    return(1);
    else
/** Segment 3 <else> **/
    return(0);
}

-----
-- TCAT/C, Release 8.2

-- END OF TCAT SEGMENT REFERENCE LISTING
-----
```

FIGURE 27 Reference Listing

The instrumentor also produces an **Instrumented Statistics** file. Statistics are organized module-by-module. The file is named *basename.i.S*. In the case of the *example.c* program it is automatically named *example.i.S*.

```
-----
--
-- TCAT/C, Release 8.2
--
-- (c) Copyright 1990 by Software Research, Inc. ALL RIGHTS
RESERVED.
--
-- INSTRUMENTATION STATISTICS
--
-- Instrumentation date: Tue May 4 15:06:02 1993
--
-----
MODULE 'main':
statements = 42
compound statements = 7

branching nodes = 10
segments instrumented = 27

conditional statements (if, switch) = 3
if statement = 1
```

```
else statement added = 1
switch statements = 2
switch statement cases = 14
default statement added = 0

iterative statements (for, while, do) = 5
for statements = 2
while statements = 3
do statements = 0

exit statement = 0
return statement = 0

MODULE 'proc_input':
statements = 22
compound statements = 6

branching nodes = 9
segments instrumented = 24

conditional statements (if, switch) = 4
if statements = 3
else statement added = 1
switch statement = 1
switch statement cases = 11
default statement added = 0

iterative statements (for, while, do) = 3
for statement = 1
while statements = 2
do statements = 0

exit statement = 0
return statements = 3
MODULE 'chk_char':
statements = 2
compound statement = 1

branching nodes = 3
segments instrumented = 3

conditional statement (if, switch) = 1
if statement = 1
else statement added = 0
switch statement = 0
switch statement case = 0
default statement added = 0

iterative statements (for, while, do) = 0
```

```
for statements = 0
while statements = 0
do statements = 0

exit statement = 0
return statements = 2

-----
-- TCAT/C, Release 8.2

-- END OF TCAT INSTRUMENTATION STATISTICS
-----
```

FIGURE 28 Instrumentation Statistics Sample

During instrumentation, a Segment Count Listing file is automatically created (*basename.i.L*). This file contains a complete count of all the modules and their logical branches in the program being tested. This file can be used with the *mkarchive* utility to create a null archive file. Please see Section 6.5, "mkarchive Utility".

Module	# Segment
main	27
proc_input	24

FIGURE 29 Segment Count Listing Sample

The Directed Graph Listing shows the relationship between nodes and logical branches. Below is the *example.c* program's Directed Graph Listing. The first two columns show the node numbers and the third column shows the branch number. To first row reads like this: Segment 1 connects nodes 1 and 2. You can also visually look at this file using the **Analyze** window's **View Source** option, or the **Xdigraph** utility.. Below is the visual representation of *chk_char.dig*'s directed graph.

```
# digraph for 'chk_char.dig' in file "example"
      1      2      1
      2      3      2
      2      3      3
```

FIGURE 30 Directed Graph Listing

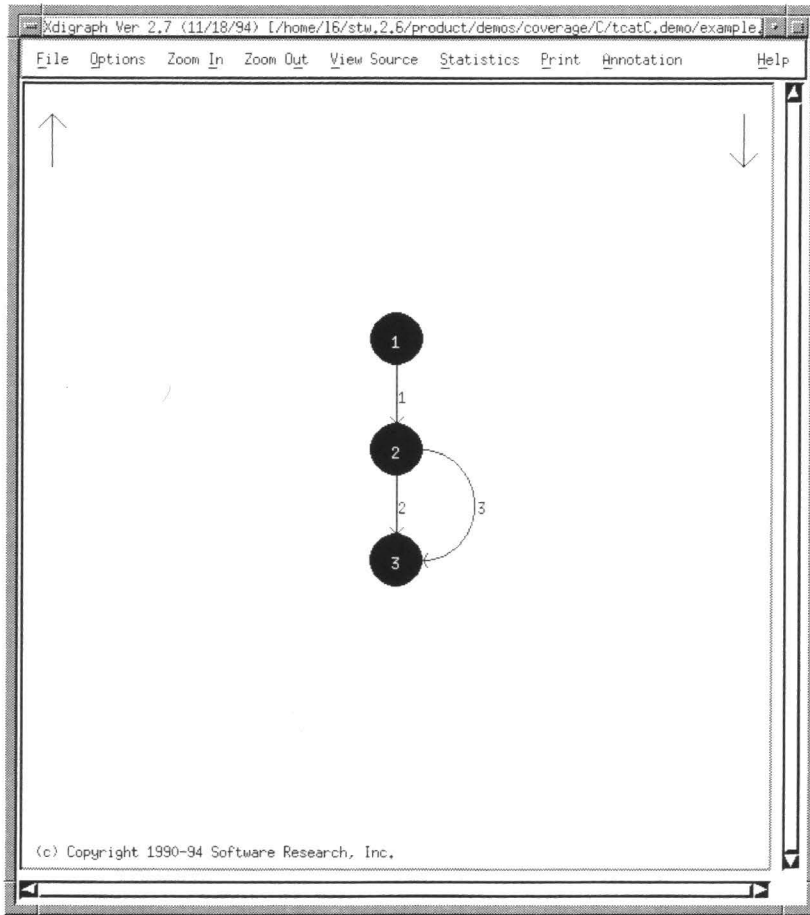


FIGURE 31 Directed Graph Display

Instrumentation errors are generally the result of typing mistakes. The instrumentor will stop at the first unrecognized character and display that line and several lines of code leading up to the point of failure. During instrumentation, the error file is automatically named *basename.i.E*. In the case of the *example.c* program it is named *example.i.E*.

 -- TCAT/C, Release 8.2

--
 -- (c) Copyright 1990 by Software Research, Inc. ALL RIGHTS
 RESERVED.

```
--  
-- ERROR LISTING  
--  
-- Instrumentation date: Tue May 4 15:06:02 1993  
--  
-----
```

FIGURE 32 Error Listing

3.2.3 Compile and Link Code

After instrumentation, you need to compile the modified program. When you compile the instrumented program, a file name *basename.i.o* is created. This file is the object code of the instrumented program. You then must link the instrumented program's object code with one of *TCAT* runtime module files. *TCAT*'s runtime modules define all the functions inserted by the instrumentor.

3.2.4 Execute Program and Generate Trace File

Once the program has been instrumented, compiled and linked with the appropriate *TCAT* runtime module, the next step is to run the program.

In your test run, *TCAT* will initially prompt you to make a comment for the current test run and to name a trace file. The trace file is where executed test information is written. If you think this takes up too much time, you can avoid this by selecting *TCAT*'s quiet runtime, *crun0.o*, which automatically defaults to the file name *Trace.trc* for a trace file.

At this point, the instrumented program will run as usual. Enter information to exercise the system control structure thoroughly.

During testing, information about branch coverage is recorded in the trace file without any work on your part. Note, however, only the latest run is stored in the trace file. Older runs are automatically stored in a file named *Archive* when you run the coverage analyzer (**cover**). This file serves as the archive library for all test runs.

3.2.5 Generate Coverage Reports

Once the test program has been executed and a trace file created, you can analyze branch coverage coverage with easy-to-read coverage reports. In general, these reports show you which logical branches have been hit or ignored during your test run.

You *first* select the kind of report you want (listed and described on the following pages) and then use the **GUI Run Coverage Analyzer** option

or the command line cover command. Depending on the type of report you select, *TCAT* will gather information from the trace file and the Archive file.

In general the reports give the following information:

1. Reports included in the current report.
2. A summary of past coverage runs.
3. Current and cumulative coverage statistics.
4. A list of logical branches that have been hit.
5. A list of logical branches that have been missed.
6. Bar charts of the frequency of execution for each branch.

These reports are useful for performance analysis and also for "hot spot" tuning.

TCAT offers the following coverage reports:

The **Cumulative** report charts branch coverage for the current test cumulatively, and for each module in the total system: its module name, number of branches, number of invokes, number of branches hit, and C1 coverage.

```

TCAT: Coverage Analyzer. [Release 8.2]
(c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.
-----+-----+-----+-----+-----+-----+-----+-----+-----+
-+
|                                     | Current Test          | Cumulative Summary   |
|                                     |-----+-----+-----|-----+-----+-----|
|                                     | No. Of               | No. Of               |
| Module                            | No. Of               | No. Of               |
| Name:                             | Segments:           | Segments             |
|-----+-----+-----+-----+-----+-----+-----+-----+-----+
| main                               | 27 | 1 | 17 | 62.96 | 2 | 17 | 62.96 |
| proc_input                         | 24 | 6 | 15 | 62.50 | 12 | 15 | 62.50 |
| chk_char                           | 3 | 6 | 2 | 66.67 | 12 | 2 | 66.67 |
|-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Totals                             | 54 | 13 | 34 | 62.96 | 26 | 34 | 62.96 |
|-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

Current test message(s) (saved in archive):
 Runtime vers 4.9, last updated 12/4/88

The **Past Test** report resembles the **Cumulative** report, but lists information from the stored archive data. It summarizes the percentage of logical branches in each module listed, giving the C1 value for each module and the program as a whole.

TCAT: Coverage Analyzer. [Release 8.2]
 (c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.

		Current Test				Cumulative Summary			
Module	Number Of	No. Of	No. Of	C1%	No. Of	No. Of	C1%		
Name:	Segments:	Invokes	Hit	Cover	Invokes	Hit	Cover		
main	27	1	18	66.67	1	18	66.67		
proc_input	24	12	14	58.33	12	14	58.33		
chk_char	3	12	2	66.67	12	2	66.67		
Totals	54	25	34	62.96	25	34	62.96		

Current test message(s) (saved in archive):
 Runtime vers 4.9, last updated 12/4/88

The **Hit** report identifies all of the logical branches which were exercised in the present and past tests. It analyzes information from both the archive and the trace file. It includes: module names, identification number for each logical branch hit so far, the number of logical branches hit, the total number of logical branches, and the resulting C1 coverage value.

TCAT: Coverage Analyzer. [Release 8.2]
 (c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.

C1 Segment Hit Report.

No.	Module Name:	Segment Coverage Status:
1	main	
	1 2 3 4 5 7 9 13 14	
	15 17 18 19 21 23 25 26 27	
2	proc_input	
	1 3 4 6 7 8 9 10 11	
	12 13 14 15 16	
3	chk_char	
	1 2	
Number of Segments Hit:		34
Total Number of Segments:		54
C1 Coverage Value:		62.96%

The **Not Hit** report indicates untested branches. This report charts, for the current test and includes the following information: branch coverage status (100% or specific branch not hit), total number of branches hit, total number of branches in the system, and C1 coverage value. You can use this information to add tests to your test suite for more comprehensive testing.

TCAT: Coverage Analyzer. [Release 8.2]
(c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.

C1 Segment Not Hit Report.

No.	Module Name:	Segment Coverage Status:
1	main	
	6 8 10 11 12 16 20 22 24	
2	proc_input	
	2 5 17 18 19 20 21 22 23	
	24	
3	chk_char	
	3	
Number of Segments Not Hit:		20
Total Number of Segments:		54
C1 Coverage Value:		62.96%

The **Newly Hit** report shows which logical branches (by module) were hit in the current execution that were not hit previously. This information gives you an assessment of the value of the most recently added test(s). This shows what the current test gained.

TCAT: Coverage Analyzer. [Release 8.2]

(c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.

C1 Segment Newly Hit Report.

No.	Module Name:	Segment Coverage Status:
1	main	
	1 2 3 4 5 7 9 13 14	
	15 17 18 19 21 23 25 26 27	
2	proc_input	
	1 3 4 6 7 8 9 10 11	
	12 13 14 15 16	
3	chk_char	
	1 2	

The **Newly Missed** report shows which branches (by module) that were not hit in the current execution that were hit previously. This information gives you an assessment of the loss of the most-recently added test(s). This shows what the current test "lost". This report is complimented by the above **Newly Hit** report.

TCAT: Coverage Analyzer. [Release 8.2]

(c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.

C1 Segment Newly Missed Report.

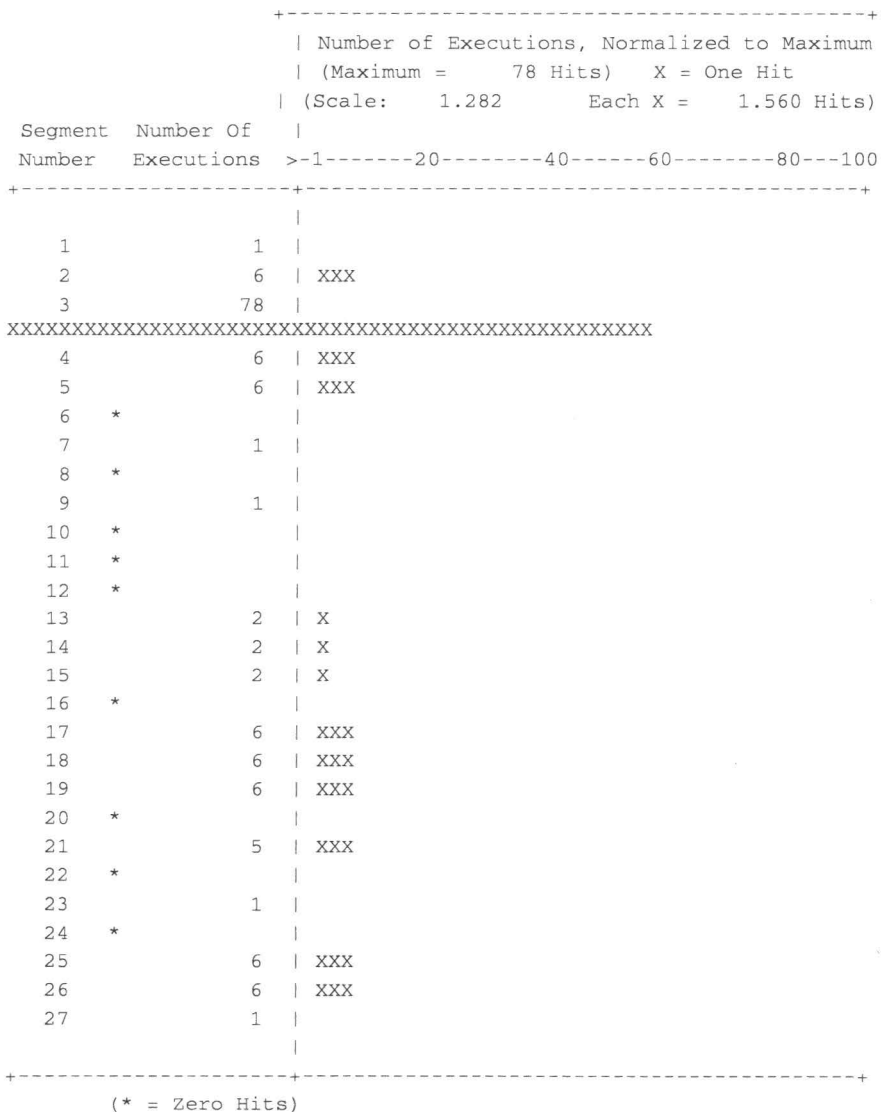
No.	Module Name:	Segment Coverage Status:
		None found.

The **Logarithmic Histogram** and the **Linear Histogram** reports demonstrate the frequency distribution of branches exercised in each module. These reports combine the current trace file and includes archive data. The **Linear Histogram** graphs a mark for each branch hit during testing; the **Logarithmic Histogram** translates this data into logarithms making the graph more readable for varying branch hit levels.

Both histograms include the following information: module name, branch numbers, number of executions, frequency distribution of exercised branches, graph scale, average hits per executed branch and module C1 value.

On the next two pages are examples of both histograms for *example.c*'s main module. Below is a **Linear Histogram** report.

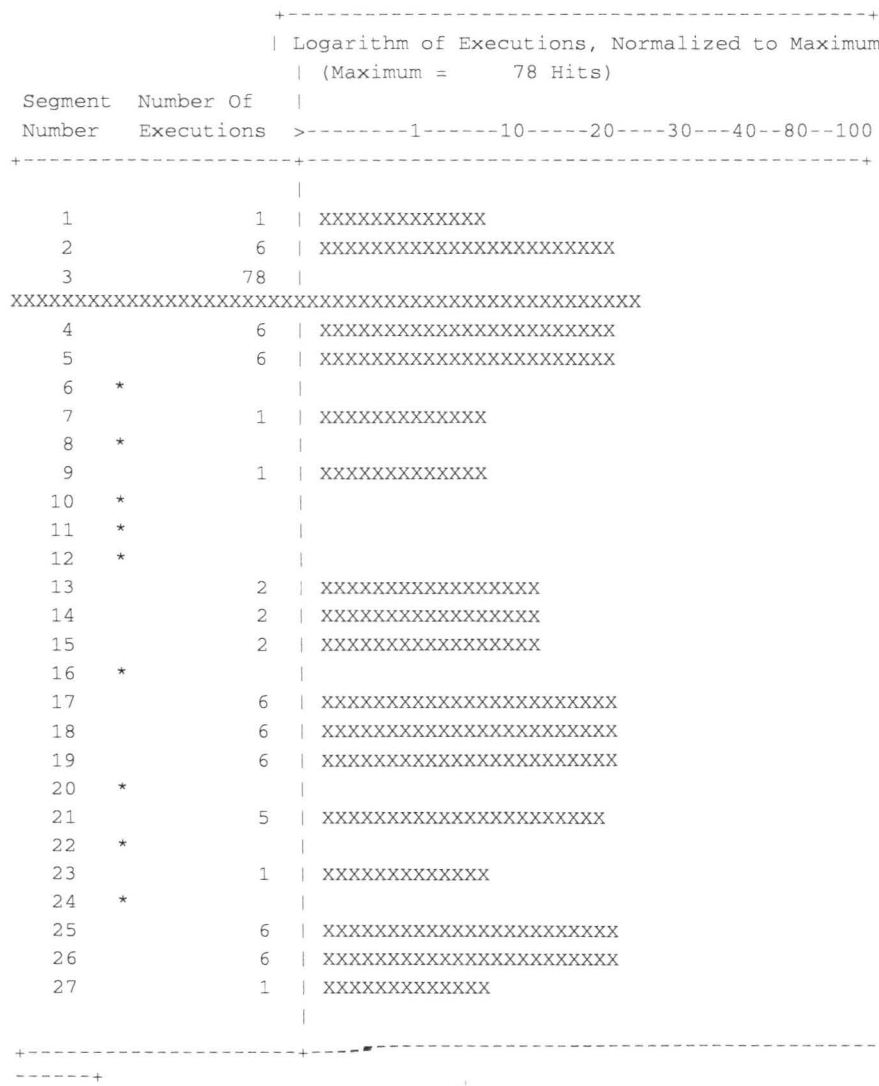
TCAT: Coverage Analyzer. [Release 8.2]
(c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.
Segment Level Histogram for Module: main



Average Hits per Executed Segment: 7.8889
 C1 Value for this Module: 66.6667

Below is a **Logarithmic Histogram** report.

TCAT: Coverage Analyzer. [Release 8.2]
 (c) Copyright 1993 by Software Research, Inc. ALL RIGHTS RESERVED.
 Segment Level Histogram for Module: main



(* = Zero Hits)

Average Hits per Executed Segment: 7.8889
C1 Value for this Module: 66.6667

The annotated **Reference** report shows the coverage level achieved for all modules that are named in the specified reference listing. If a module is tested but the name is not found in the supplied reference listing, then that coverage is not reported. Similarly, if a name appears in the reference listing but is not found in the archive or trace file, no coverage will be reported.

On the following page is an example of reference listing report. Take note of the following:

- Point A Marks the reference listing file name.
- Point B Shows the C1 coverage for the module.
- Point C Indicates how many times a branches has been hit.
- Point D Refers to an unexecuted logical branch.

```
TCAT: Coverage Analyzer. [Release 8.2]
(c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RESERVED.
+-----+
TCAT Coverage on Reference Listing Report, based on file [|exam-
ple.i.A|].      A
+-----+

(Coverage values for all tests processed are reported in left-hand
column.
"*****" indicates not hits on corresponding segment. Extra names
not
part of this listing but in the Archive file are ignored.)

-----
--
-- TCAT, Release 8.
--
-- (c) Copyright 1990 by Software Research, Inc. ALL RIGHTS
RESERVED.
--
-- SEGMENT REFERENCE LISTING
--
-- Instrumentation date: Sat Jun 16 15:53:06 1990
--
-- Separate modules and segment definitions for each module are
```

```

-- indicated in this commented version of the supplied source file.
-----
externstruct _iobuf {
int_cnt;
unsigned char*_ptr;
unsigned char*_base;
char_flag;
char_file;
} _iob[60];
externstruct _iobuf*fopen(), *fdopen(), *freopen(), *popen(), *tmp-
file();
externchar*fgets(), *gets(), *ctermid(), *userid();
externchar*tempnam(), *tmpnam();
externvoidrewind(), setbuf();
externlongftell();
externunsigned char*_bufendtab[];
externchar_ctype[];
char menu[13][79] = {
"SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
"\n",
"    What type of food would you like?\n",
"\n",
"    1    American 50s    \n",
"    2    Chinese    - Hunan Style \n",
"    3    Chinese    - Seafood Oriented \n",
"    4    Chinese    - Conventional Style \n",
"    5    Danish        \n",
"    6    French        \n",
"    7    Italian       \n",
"    8    Japanese     \n",
"\n\n"
};
int char_index;
main(argc,argv)
intargc;
char*argv[];
{
int i, choice, c;
char str[79], answer;
int ask, repeat;

+-----+
|C1 = 66.67|/** Module main **/      B
+-----+
/* First Segment*/
++int proc_input();
|1|/** Segment 1 <> **/      C
+++
c = 3;

```



```
repeat = 1;
while(repeat) {
6/** Segment 2 <start while> **/
printf("\n\n\n");
for(i = 0; i < 13; i++)
78/** Segment 3 <start for> **/
printf("%s", menu[i]);
6/** Segment 4 <end for> **/
gets(str);
printf("\n");
while(choice = proc_input(str)) {
6/** Segment 5 <start while> **/
switch(choice) {
case 1:
+-----+
|*****/** Segment 6 <case alt> **/      D
+-----+
printf("\tFog City Diner  1300 Battery   982-2000 \n");
break;
case 2:
1/** Segment 7 <case alt> **/
printf("\tHunan Village Restaurant  839 Kearney   956-7868 \n");
break;
case 3:
*****/** Segment 8 <case alt> **/
printf("\tOcean Restaurant 726 Clement   221-3351 \n");
break;
case 4:
1/** Segment 9 <case alt> **/
printf("\tYet Wah 1829 Clement   387-8056 \n");
break;
case 5:
*****/** Segment 10 <case alt> **/
printf("\tEiners Danish Restaurant 1901 Clement   386-9860 \n");
break;
case 6:
*****/** Segment 11 <case alt> **/
printf("\tChateau Suzanne 1449 Lombard   771-9326 \n");
break;
case 7:
*****/** Segment 12 <case alt> **/
printf("\tGrifone Ristorante 1609 Powell   397-8458 \n");
break;
case 8:
2/** Segment 13 <case alt> **/
printf("\tFlints Barbeque 4450 Shattuck, Oakland \n");
break;
default:
2/** Segment 14 <case alt> **/
```

```

if(choice != -1)
2/** Segment 15 <if> **/
printf("\t>>> %d: not a valid choice.\n", choice);
****/** Segment 16 <implied else> **/
break;
}
}
6/** Segment 17 <end while> **/
for(ask = 1; ask; ) {
+--+
|6|/** Segment 18 <start for> **/    C
+--+
printf("\n\tDo you want to run it again? ");
while((answer = ( --((&_iob[0])->_cnt >= 0 ? (0xff & (int)
*((&_iob[0])->_ptr++)) : _filbuf((&_iob[0])) )) != '\n') {
6/** Segment 19 <start while> **/
switch(answer) {
case 'Y':
****/** Segment 20 <case alt> **/
case 'y':
5/** Segment 21 <case alt> **/
ask = 0;
char_index = 0;
break;
case 'N':
****/** Segment 22 <case alt> **/
case 'n':
1/** Segment 23 <case alt> **/
ask = 0;
repeat = 0;
break;
default:
+-----+
|****|/** Segment 24 <case alt> **/    D
+-----+
break;
}
}
6/** Segment 25 <end while> **/
}
6/** Segment 26 <end for> **/
}
1/** Segment 27 <end while> **/
}
int proc_input(in_str)
char *in_str;
{
int tempresult = 0;
char bad_str[80], *bad_input;

```

```
+-----+
|C1 = 58.33|/** Module proc_input **/      B
+-----+
/* First Segment*/
  int got_first = 0;
12/** Segment 1 <> **/
bad_input = bad_str;
while(((_ctype+1)[in_str[char_index]]&010))
****/** Segment 2 <start while> **/
char_index++;
+--+
|12|/** Segment 3 <end while> **/      C
+--+
for( ; char_index <= strlen(in_str); char_index++) {
19/** Segment 4 <start for> **/
switch(in_str[char_index]) {
case '0':
+-----+
|****|/** Segment 5 <case alt> **/      D
+-----+
case '1':
1/** Segment 6 <case alt> **/
case '2':
3/** Segment 7 <case alt> **/
case '3':
3/** Segment 8 <case alt> **/
case '4':
4/** Segment 9 <case alt> **/
case '5':
4/** Segment 10 <case alt> **/
case '6':
4/** Segment 11 <case alt> **/
case '7':
4/** Segment 12 <case alt> **/
case '8':
6/** Segment 13 <case alt> **/
case '9':
7/** Segment 14 <case alt> **/

tempresult = tempresult * 10 + (in_str[char_index] - '0');
got_first = 1;
break;
default:
12/** Segment 15 <case alt> **/
if(chk_char(in_str[char_index])) {
12/** Segment 16 <if> **/
return(tempresult);
}
}
```

```

else {
*****/** Segment 17 <else> **/
if(char_index > 0 && got_first)
*****/** Segment 18 <if> **/
char_index--;
*****/** Segment 19 <implied else> **/
while(char_index <= strlen(in_str)) {
*****/** Segment 20 <start while> **/
if(chk_char(in_str[char_index]))
*****/** Segment 21 <if> **/
break;
else
*****/** Segment 22 <else> **/
*bad_input++ = in_str[char_index];
char_index++;
}
*****/** Segment 23 <end while> **/
*bad_input = '\0';
printf("\t>>> bad input: %s\n", bad_str);
char_index++;
return(-1);
}
}
}
*****/** Segment 24 <end for> **/
return(0);
}
int chk_char(ch)
char ch;

+-----+
|C1 = 66.67|** Module chk_char **/      B
+-----+
/* First Segment*/
{
+--+
|12|** Segment 1 <> **/                C
+--+
if(((ctype+1)[ch]&010) || ch == '\0')
12/** Segment 2 <if> **/
return(1);
else
+-----+
|****|** Segment 3 <else> **/          D
+-----+
return(0);
}

-----
-- TCAT/C, Release 8.2

```

-- END OF TCAT SEGMENT REFERENCE LISTING

3.3 Conclusion

From this chapter you should have learned the basic steps needed to use *TCAT*: instrument and compile the program, execute the program and generate a trace file and generate reports. In the examples shown throughout this chapter, C1 coverage was only around 63 percent. Ideally, you want to try for 85 percent coverage. In the case of this example, you would re-run the program and execute new tests to achieve higher coverage.

TCAT can do a number of things for you: manage your system testing more objectively and effectively. It also finds latent software errors before your customers do. Finally, *TCAT* demonstrates when testing is complete with its easy-to-read reports.



GUI Operation

This chapter covers the basic X Window System graphical user interface (GUI) operations of *TCAT*. It demonstrates using *TCAT* from the OSF/Motif X Window System.

LEVEL: If you are an advanced X Window System *TCAT* user, you may skip this chapter, which is intended for beginning and intermediate users.

4.1 User Interface

If you are familiar with the OSF /Motif style graphical user interface, you can go on to the next section. This section demonstrates using file selection dialog boxes, help menus, message dialog boxes, and pull-down menus.

4.1.1 File Selection Box

The **Instrument**, **Execute** and **Analyze** windows use file selection dialog boxes, where you can select a new or existing file name.

Refer to the next figure for each of the dialog box's components:

Filter entry box	Specifies a directory mask. When you click the Filter push button, the directory mask is used to filter files or directories that match this mask (or pattern).
Directories	Lists directories in path defined in the Filter entry box.
Files	Lists files in path defined in the Filter entry box.
Scroll Bars	Move up/down and side/side in the Directories and Files selection windows. You use them to search for the appropriate directory or file.
Selection entry box	Selects and enters the file name.

Use the three push buttons at the bottom of the dialog box to issue commands:

OK	Accepts the file in the Selection entry box as the new file or the file to be opened and then exits the dialog box.
-----------	--

- Filter** Applies the pattern you specified in the **Filter** entry box. It lists the directories and files that match that pattern.
- Cancel** Cancels any selections made and then exits the dialog box. No file is selected as a result.

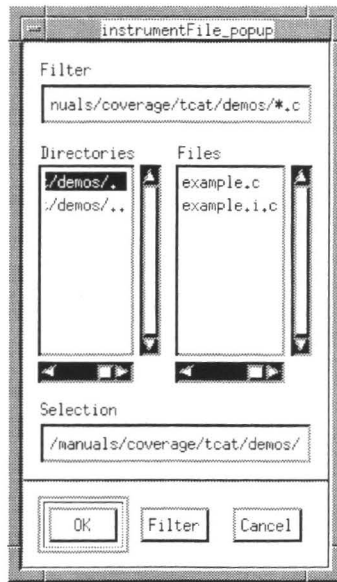


FIGURE 33 Using a File Selection Dialog Box

To use a file selection dialog box, follow these steps:

1. You can restrict the file selection operation to a named region (directory path) by typing in a directory path name in the **Filter** entry box or by clicking on a path name in the **Directories** selection window. Then click on the **Filter** push button.
2. Select a file by clicking on an already existing file you want to overwrite in the **Files** selection window or type in a new file name in the **Selection** entry box, with no limit on character length.
3. To select a file name, do one of these three things:

- Double click on the file in the File selection window, Highlight the file in the File selection window, or
- type in the file name in the **Selection** entry box and click **OK**, or
- Highlight or type in the file name and press the <ENTER> key.

4.1.2 Help Boxes

TCAT provides on-line help for its **Main**, **Instrument**, **Execute**, and **Analyze** windows. This on-line help will automatically bring up the text corresponding to where you invoke it at. In other words, if you invoke it at the Main window, the **Help** window will automatically display information pertinent to the Main window. Here's how to use a help frame:

1. Once it is invoked, the text should correspond to the window from which it was invoked.
2. You can use the scroll bars to move up/down and side/side.
3. If you don't see what you need, you can search for specific text:
 - Click on the **Action** pull-down menu and select **Search**.
 - A dialog box (shown below) pops up.
 - Type in the pattern you want to search for and then click on **OK** or press the <ENTER> key.
 - If the pattern is found, the help frame will automatically scroll to the location of the pattern.
4. If you select another **Help** option from another window, while the current one is displayed, the **Help** window will automatically scroll to the context of the new window.
5. To exit, click on **Quit**.

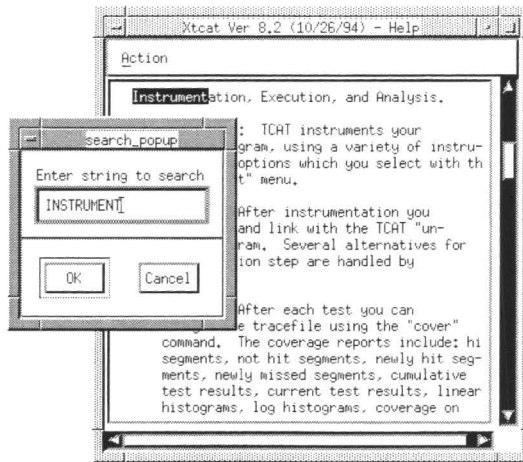


FIGURE 34 Using the Help Dialog Box

4.1.3 Message Boxes

Pop-up message dialog boxes have three purposes:

1. They display warnings and error information.
2. They ask you to verify that you want to perform a task.
3. They ask to enter a command.

To remove a message box after you have read it or to tell *TCAT* to go ahead with a command, click the **OK** push button. If you want to cancel a command, click the **Cancel** push button.

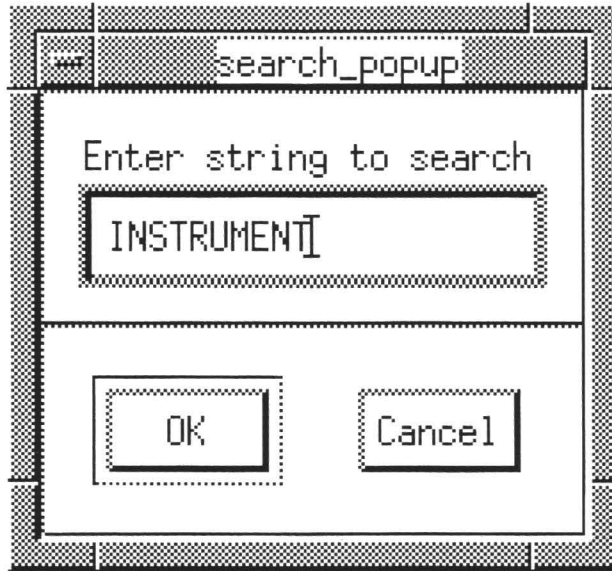


FIGURE 35 Using a Dialog Box

4.1.4 Option Menus

The **Instrument**, **Execute** and **Analyze** windows use an option menu. An option menu includes selections from a list. Usually, only the default menu option is visible. To use an option menu, follow these steps:

1. Click on the option menu.
2. After clicking on the menu, the list of choices are visible.
3. Drag the mouse to the menu option you want.
4. Let go of the mouse.
5. The new menu option should be visible, indicating that it is activated.

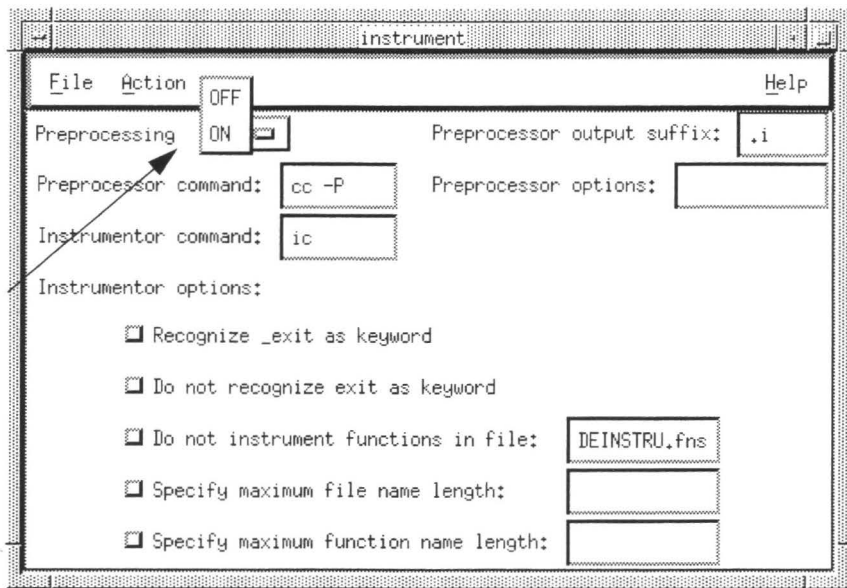


FIGURE 36 Using an Option Menu

Pull-down menus are located within the menu bar. They often contain several options. To use pull-down menus and their options, follow these steps:

1. Move the mouse pointer to the menu bar and over the menu containing the item.
2. Hold the left mouse button down. This displays the items on the menu.
3. While holding down the left mouse button, slide the mouse pointer to the menu item you want to select. The menu item is highlighted in reverse shadow.

Three dots at the right of the menu item indicates that selecting the item will bring up a pop-up window.

An arrow to the right of the menu item indicates that the item is a submenu (or cascading menu).

To display the submenu, slide the mouse pointer over the arrow. You can then select an item on the submenu.

4. Release the mouse button while the desired item is highlighted to activate the command. To the function exit without selecting anything, simply drag the mouse pointer off the menu before releasing the mouse button to not activate anything.

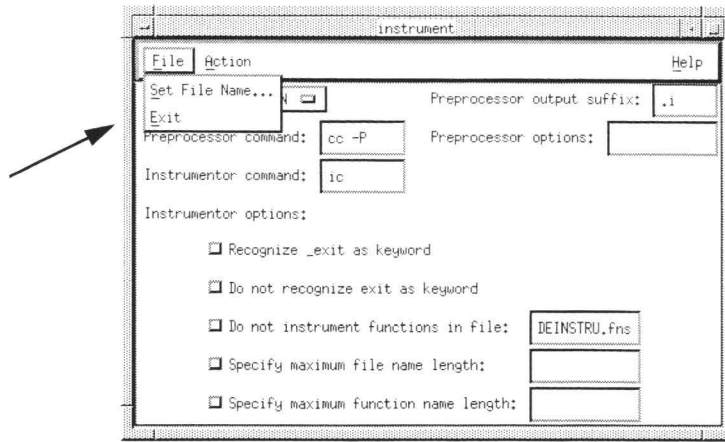


FIGURE 37 Using a Pull-down Menu

4.2 Invoking TCAT

To start *TCAT* from your working directory, type this command:

```
Stcat
```

The **Main** window (shown below) pops up.



FIGURE 38 Invoking the Main Window

You can also invoke *TCAT* through the **STW** menu. First, type

```
stw
```

1. The **STW** window (shown below) pops up.
2. Click on the **Coverage** activation button.
3. The **STW/Coverage** window pops up.
4. Click on *TCAT*. *TCAT*'s **Main** window pops up.

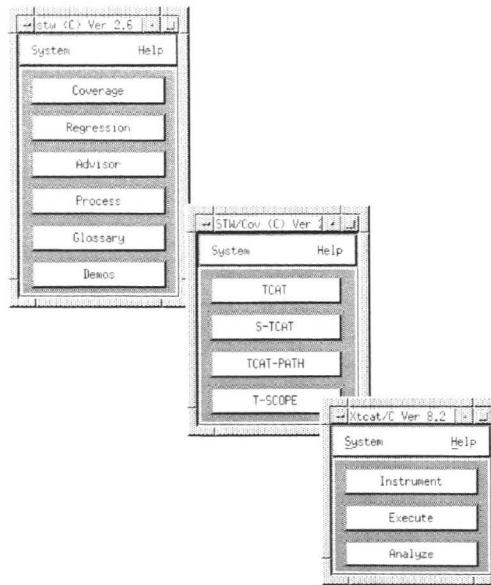


FIGURE 39 Invoking TCAT from the STW Tool Suite

4.2.1 Selecting Main Window Options

The **Main** window has four push buttons that allow you to perform all of *TCAT*'s operations, including, instrumenting your application, compiling, linking object code, executing the program, generating a trace file and looking at coverage reports or source code.

Instrument	For preprocessing and instrumenting your application. (See Section 5.3)
Execute	For compiling the instrumented version of your program, linking the program's object code to <i>TCAT</i> 's

object modules, and running the application. (See Section 5.4)

Analyze For generating coverage reports and visually looking at the source code. (See Section 5.5 and the accompanying documentation on the **Xdigraph** utility.)

The following sections deal specifically with their usage.

4.2.2 Exiting the Main Window

The Exit option allows you to close *TCAT*.

Here's how:

1. Click on the **System** pull-down menu.
2. Drag the mouse to **Exit**, and then let go of the mouse button. *TCAT* exits.

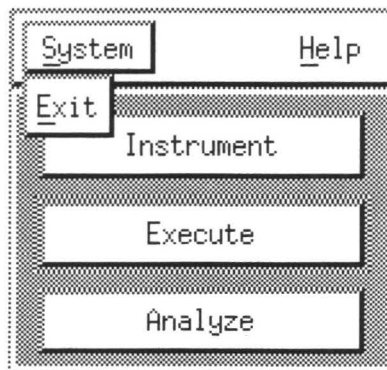


FIGURE 40 Exiting the Main Window
4.3 Instrumenting

To analyze your test coverage you must first preprocess your application for syntax errors and then instrument it. During instrumentation, *TCAT* modifies your application by placing special markers (function calls) at every logical branch in each program module. These markers are later tracked and counted by *TCAT* during your application's execution. This is how coverage is obtained.

To begin the instrumentation process, invoke the **Instrument** window by clicking on the **Instrument** activation button. The window below pops up.

NOTE: If you are instrumenting with make files, please refer to Section 4.5.

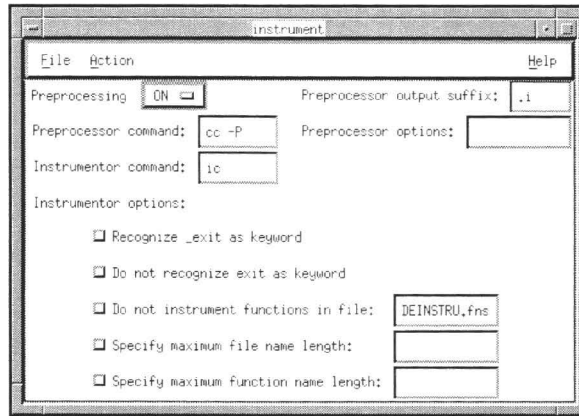


FIGURE 41 Invoking the Instrument Window

4.3.1 Selecting the Application Name

You must first select an already existing program to instrument. To select a file name:

1. Click on the **File** pull-down menu.
2. Select **Set File Name**.
3. A file selection dialog box like the one below pops up.
4. Select an existing program name, *basename*.
5. Select a file name by clicking on an already-existing program in the **Files** selection window or typing in the file name in the **Selection** entry box.

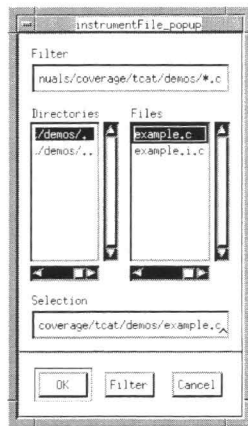


FIGURE 42 Selecting the Program File Name

4.3.2 Setting Options

After selecting the application file name, you may adjust the Instrument window's options. Listed next are the options and how to use them. These instructions will also tell you how to change the defaults. If you want a more permanent change, you can change any of the defaults in the *.Xdefaults* file (see Chapter 8). For complete definitions on the options' functions, please refer to Chapter 5.

4.3.2.1 Preprocessing Option Menu

Most often you must check your program for syntax errors by preprocessing. In these cases, you will simply leave the default **ON** switch on. Sometimes, however, you may already know there are no syntax errors and want to skip the preprocessing step. In these cases, you will want to activate the **OFF** switch. Here's how:

1. Click on the **Preprocessing** menu.
2. Select the **OFF** switch.
3. The accompanying **Preprocessor command**, **Preprocessor output suffix** and **Preprocessor options** options gray out, becoming inactive.

4.3.2.2 Preprocessor output suffix

When you preprocess a program, normally the file name will change from *basename* to *basename.i*, with an *i* suffix. If you want a different suffix:

1. Click inside the corresponding specification region.
2. When the cursor appears, you can type in the desired suffix.

4.3.2.3 Preprocessor command

The preprocessor command is defaulted to `cc -P`, a standard UNIX pre-processing command. If you want to change it:

1. Click inside the corresponding specification region.
2. When the cursor appears, you can edit.

4.3.2.4 Preprocessor options

If you want to add additional compiler options for preprocessing:

1. Click inside the corresponding specification region.
2. When the cursor appears, you can edit.

4.3.2.5 Instrumentor command

The instrumentor command is defaulted to `ic`. This is the command that instruments your C program. To change it:

1. Click inside the corresponding specification region.
2. When the cursor appears, you can edit.

4.3.2.6 Instrumentor options

This option provides several check buttons from which you can select. These options will effect the instrumentation process in several ways. You can select any of the following check buttons (by clicking once in the corresponding button). A check button is turned on if it darkened. If it is hollowed, then it is turned off.

- **Recognize `_exit` as keyword** button. You turn this option on if you want the instrumentor command (`ic`) to recognize the keyword `exit` in your program.
- **Do not recognize `_exit` as keyword** button. You turn this option if you do not want the instrumentor command (`ic`) to recognize the keyword `exit` in your program.
- **Do not instrument functions in file** button. Use this option to selectively de-instrument individual C functions, or modules. *TCAT* will disregard these functions when they are found.

This option can effectively ignore entire modules from instrumentation. You should use this option when you don't want a particular module's logical branches marked during instrumentation.

The default file is set to *DEINSTRU.fns*. If you want to de-instrument certain functions, simply put the names of those functions you want to de-instrument in this file. If you want to change the name of the default file name, click inside the specification region and begin editing when the cursor appears.

NOTE: You can also de-instrument parts of your code by placing directives in your source code file. Please refer to Section 6.3.2 for further information.

- **Specify maximum file name length** button. Use this option when your system has a limit on the amount of characters a file name can have. If the length exceeds the value, then the instrumentor output will be redirected to files named *Temp.i.?*. (See Section 4.5.4 for a listing of the different kinds of instrumentor output).

Type in the amount of characters in the accompanying specification region.

- **Specify maximum function name length** button. Use this option when your system has a limit on the amount of characters a name can have. If the length exceeds the value, then the instrumentor will recognize only the first value characters of the function name. For instance, a value of 5 will recognize only the first five characters of a module as distinct. Characters beyond that point will not be recognized for function name purposes.

4.3.3 Preprocessing Your Program

After selecting the **Instrument** window's options, you are ready to preprocess your program. Prior to instrumenting, it is often necessary to preprocess your program for syntax errors. Here's how:

1. Click on the **Action** pull-down menu.
2. Select **Preprocess**.
3. The mouse pointer turns into a wristwatch symbol and the Instrument window's options gray out until preprocessing is complete. This signifies a time-out period in which the *TCAT* is completely inactive until preprocessing is complete.

NOTE: If you turned the preprocessing **OFF**, you do not have to preprocess.

Preprocessing Results

Preprocessing checks your program for syntax errors. If any are found, messages are displayed in the invocation window.

When preprocessing is complete, *TCAT* writes its results to a file named *basename.i*, where *basename* is the name of your program and *i* indicates it as a preprocessed file.

4.3.4 Instrumenting Your Program

After preprocessing your program, you are ready to instrument your program. During this phase, *TCAT* will automatically insert function calls at each logical branch. This marking is important. Later when you run your application, you will be trying to hit these markers with your planned test suite. This information is then written to a trace file, where you can obtain coverage reports.

Here's how to instrument:

1. Click on the **Action** pull-down menu.
2. Select **Instrument**.
3. The mouse pointer turns into a wristwatch symbol and the **Instrument** window's options gray out until preprocessing is complete. This signifies a time-out period in which the *TCAT* is completely inactive until instrumentation is complete.
4. When instrumentation is complete and no errors are found, the following message appears in the invocation window:

```
---> TCAT analysis of 'basename ' complete, no errors <---
```
5. If an error is found, it will appear in the invocation window.

NOTE: If you used any of the **Instrumentor** options, instrumentation will be affected accordingly.

Instrumenting Results

Instrumentation produces the following files:

- *basename. i.c* -- an instrumented version of your C program, *basename*.
- *basename. i.A* -- a Reference Listing, which has the logical branches marked as Segment 1, Segment 2...
- *basename. i.S* -- an Instrumented Statistics file, where various kinds of statistics are listed for each module, including the number of statements, logical branches, conditional statements, etc.
- *basename. i.L* -- a Segment Count Listing file, which contains a complete count of all the modules and their logical branches in the program being tested.
- *modulename. dig* -- a Directed Graph Listing file for each module, which reports the logical branch relationship between nodes. You can also visually look at a module's directed graph using *TCAT*'s **Xdigraph** utility (see Chapter 22, "**Xdigraph** Utility").

- *basename.i.E* -- an Error Listing file, which contains all the errors found during instrumentation.

To look at samples of the above files, please refer to Section 4.2.

4.3.5 Exiting the Instrument Window

The **Exit** option allows you to close the Instrument window. Here's how:

1. Click on the **System** pull-down menu.
2. Drag the mouse to **Exit** and then let go of the mouse button. The **Instrument** window exits.

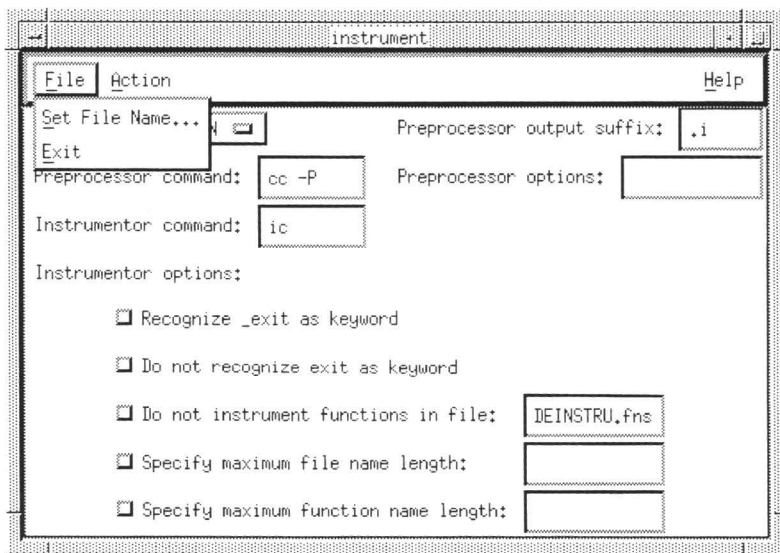


FIGURE 43 Exiting the Instrument Window

4.4 Running Your Program

After instrumenting your source program, you need to compile the instrumented version of your program, link the program's object code with *TCAT*'s runtime object modules, and run your application.

As you know, instrumentation inserts function calls at each logical branch/call-pair. When you eventually run the program, you will be trying to "hit" these function calls. In order for *TCAT* to understand the meaning of the instrumented program's object code, you must link the code to a supplied runtime object module. This runtime module will interpret the object code's instructions, creating an executable. After linking, you can run your program (see Section 4.5.6).

This is all accomplished using the **Execute** window. This section demonstrates the **Execute** window.

4.4.1 Invoking the Execute Window

Invoke the **Execute** window from the **Main** window. Simply click on the **Execute** button. The window below pops up.

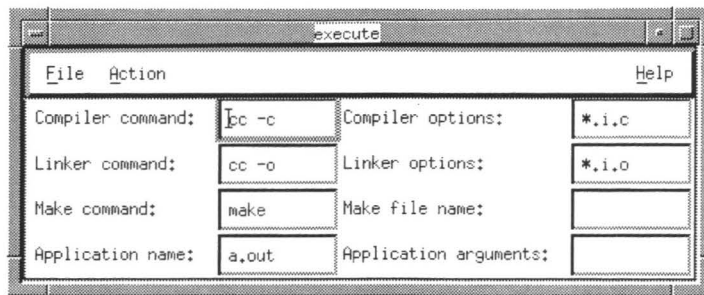


FIGURE 44 Invoking the Execute Window

4.4.2 Setting Options

When using the **Execute** window, you may want to adjust the options. Listed next are the options and their default settings. For complete definitions on the options' functions, please refer to the Chapter 5.

To change any of the default setting for the following options: position the mouse pointer so it in the specification region and then click the mouse button. A cursor will appear and you can then edit:

- **Compiler command & Compiler options.** These two options form the standard command to compile instrumented files. The **Compiler command** default is set to `cc -c` and the **Compiler options** is set to `*.i.c`. `cc -c` is the standard compiling command and `*.i.c` represents instrumented files.
- **Linker Command & Linker options.** These two options form the standard command to link the program object code files with one of *TCAT*'s object modules. The **Linker Command** default is set to `cc -o` and **Linker options** is set to `*i.o`. `cc -o. i *i.o` represents the input object code files, created during compilation.
- **Make Command.** This option invokes the **make** utility. The default is set to **make**.
- **Make file name.** This option names the 'make' file. No default is set.
- **Application name.** This names the instrumented executable. The executable is the result of linking.
- **Application arguments.** This option lists arguments or switches for the application.

NOTE: All defaults can also be changed by manually editing the *.Xdefaults* file. Please refer to Chapter 8 for further information.

4.4.3 Compiling the Instrumented Program

You are now ready to compile your instrumented program. Follow these steps:

1. Click on the **Action** pull-down menu.
2. Select **Compile**.
3. The mouse pointer turns into a wristwatch symbol and the **Execute** window's options gray out until compilation is complete.

Compilation Results

Compiling checks your instrumented program for syntax errors. If any are found, messages are displayed in the invocation window.

When compilation is complete, *TCAT* automatically writes object code found in the instrumented program to a file named *basename.i.o*, where *basename* is the name of your program and *i.o* signifies the instrumented program's object code file. Eventually this file will be linked with one of *TCAT*'s runtime object modules.

4.4.4 Selecting a Runtime Object Module

Before you link, you must specify the *TCAT* runtime object module. Each runtime routine can change the behavior and performance of the instrumented system when it is run. Below are standard routines available from *TCAT*. *TCAT* also offers several more. For more information on these, please refer to Chapter 7. Here's how to select a runtime routine:

1. Click on the **File** pull-down menu.
2. Select **Set Runtime Obj Module**.
3. A file selection dialog box like the one on the next page pops up. In the **Files** selection window, there are three runtime object modules from which to choose from:
 - *crun0.o* or quiet runtime. There is no internal processing or buffering. The trace file is the full, unedited trace of program execution. There is no prompting for trace file name at the start of your instrumented system's test run, so the trace file name is automatically defaulted to *Trace.trc*.
 - *crun1.o* This is the same as *crun0.o*, except it prompts you to describe the test and the name of the trace file. There is no processing or buffering. The trace file is the full, unedited trace of program execution. This is the most commonly-used object module.

- *cruna.o*. This runtime object module is designed for analysis of system calls such as spawn system command of C. A trace file is produced for parent and child processes.
4. Select a runtime routine by clicking on one of them in the **Files** selection window or typing in the name of the object module in the **Selection** entry box.

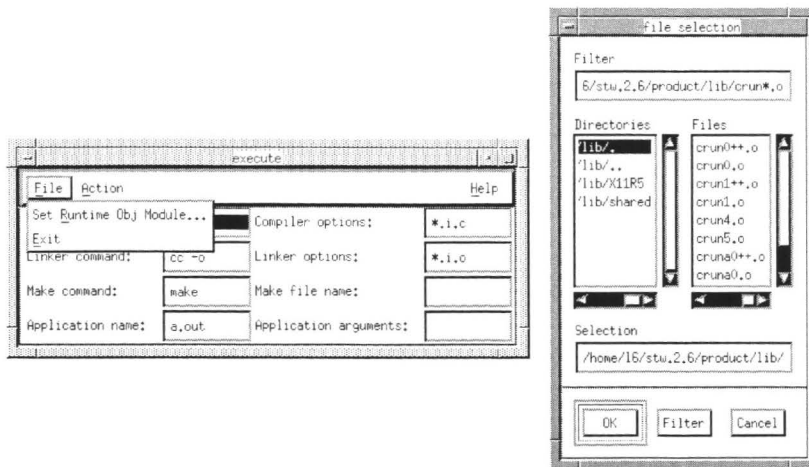


FIGURE 45 Selecting the Runtime Object Module

4.4.5 Linking

Now, you are ready to link the program's object code to the object module you selected. To link:

1. Click on the **Action** pull-down menu.
2. Select **Link**.
3. The mouse pointer turns into a wristwatch symbol and all the options gray out until the object modules are linked.

Linking Results

After linking object files, an executable of the instrumented application is created. The executable is defaulted to *a.out*.

4.4.6 Running Your Application

The next step is to run your instrumented program and track which logical branches have been exercised by the test data you supply. *TCAT* senses when segments are hit by monitoring the markers during instrumentation and by accumulating the results in a trace file. The trace file becomes the basis for all subsequent coverage reports.

To run your application:

1. Click on the **Action** pull-down menu.
2. Select **Run Application**.
3. The mouse pointer turns into a wristwatch symbol and all the options gray out until you are finished running your application.
4. If you using the *crun1.o* or *cruna.o* runtime object modules, the invocation window then prompts you:

Trace Descriptor:

Type in a description of the test run. Be as descriptive as you feel is necessary. You can enter up to 80 characters of text in your message. This message will be recorded in the trace file and used in coverage reports. If you choose to enter no descriptive text, just press the **RETURN** key.

5. If you using the *crun1.o* or *cruna.o* runtime object modules, the invocation window prompts you:

Name of trace file [default is Trace.trc]:

Type in any name. The system put the trace information under the name you specify. You can also save trace information to the default trace file name, *Trace.trc*. To do this, press the **RETURN** key.

The trace file description and trace file name are useful in keeping track of different test runs. Consistent, clear naming conventions are useful in organizing different groups of results. A recommended practice is to identify trace files with the file name extension *trc*.

If you are using the *crun0.o* runtime routine, then you will not be prompted with the questions in 4 and 5. The trace file name is automatically defaulted to *Trace.trc*.

6. Run your program as you normally would, making sure to exercise your test suite as thoroughly as possible.

Running Results

After exercising your test suite, all the test trace information is written to a trace file. From this file, coverage reports can be obtained.

4.4.7 Exiting the Execute Window

The Exit option allows you to close the **Execute** window. Here's how:

1. Click on the **File** pull-down menu.
2. Select **Exit**. The **Execute** window exits.

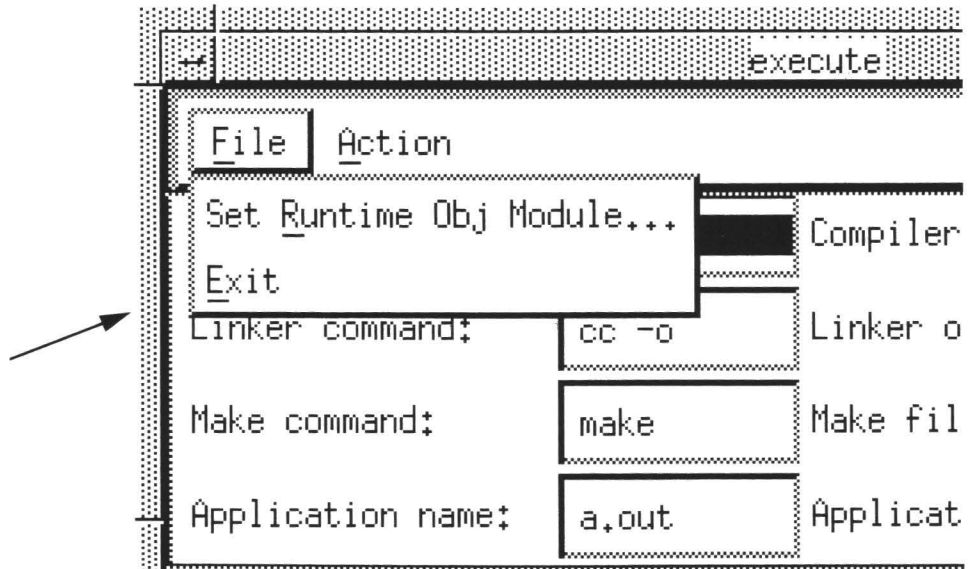


FIGURE 46 Exiting the Execute Window

4.5 Using make Files

Most often, *TCAT* will be used to develop test suites for systems that are created with make files. Make files cut the time of constructing systems, by automating the various steps necessary to build the system, including preprocessing, instrumenting, compiling and linking. All of these steps can be written in the make file.

4.5.1 Preprocessing, Instrumenting, Compiling

Fortunately, it is possible to add a few statements to most make files to enable them to make an instrumented version of the system. The modifications fall into one category: **cc** for most UNIX compilers.

If the make file explicitly mentions the C compiler with a **cc** command (for example), it is possible to add the **ic** command and an extra **cc** command for preprocessing, instrumenting and compiling, causing the make script to instrument and compile the C files in question.

This section will discuss how to use *TCAT* and make files. Please refer to Section 6.6.1 for more information on make file commands.

Make file lines such as:

```
sample.o:sample.c
cc -c sample.c
```

would be changed to:

```
sample.o: sample.c
cc -P $(CFLAGS) sample.c
ic sample.i
cc -c $(CFLAGS) sample.i.c
mv sample.i.o sample.o
```

The other situation is where the compiler is not explicitly mentioned, but given as a "built-in" rule. You can add the following "built-in" rule:

```
cc -P $(CFLAGS) *.c
ic *.i
cc -c $(CFLAGS) *.i.c
mv *.i.o *.o
```

4.5.2 Linking Object Modules

You can also link object modules by adding one of *TCAT*'s supplied object modules to the link statement. Below is a standard link statement:

```
sample: $(Objects)
rm -f sample
cc $(Objects) $(LDFLAGS) $(Lextras) -o sample
```

You would add one of the supplied object modules, as shown below (the object modules are shown in regular text):

```
sample: $(Objects) crun1.o
rm -f sample
cc $(Objects) crun1.o $(LDFLAGS) $(Lextras) -o sample
```

At this point, your make file has accomplished the preprocessing, instrumenting, compiling, and linking steps. All you need to do is run the make file (see Section 4.5.4).

4.5.3 Example make Files

The make file below shows a typical UNIX/XENIX make file before modification.

```
#####
####
##
## S A M P L E   M A K E   F I L E
##
## Make file example, no instrumentation.
##
## UNIX, XENIX
##
#####
####
# Uses make's knowledge of lex, yacc, cc.
#####
####

CCextras =
CFLAGS = -s ${CCextras} -DXENIX
YFLAGS = -d
LDFLAGS = -i -ly -ll
LFLAGS = -v
Lextras =
Objects = sample.o sample.y.o sample.l.o tree.o init.o error.o
dotest.o log.o \
ui.o premain.o preprocy.o preprocl.o pretree.o help.o license.o
Sources = sample.c sample.y.c sample.l.c tree.c init.c error.c
dotest.c log.c \
ui.c premain.c preprocy.c preprocl.c pretree.c sample.h \
typedef.h error.h y.tab.h preproc.h help.c license.c license.h
# UNIX version. Compiles and links.
sample: $(Objects)
rm -f sample
cc $(Objects) $(LDFLAGS) $(Lextras) -o sample
#
sample.y.c: sample.y
yacc $(YFLAGS) sample.y
mv y.tab.c sample.y.c
cp y.tab.h ytab.h
#
sample.l.c: sample.l
lex $(LFLAGS) sample.l
mv lex.yy.c sample.l.c
#
preprocy.c: preprocy.y
yacc $(YFLAGS) preprocy.y
cat y.tab.c | sed -e 's/yy/xx/g' > preprocy.c
cat y.tab.h | sed -e 's/yy/xx/g' > pretab.h
```



```

rm y.tab.c
#
preprocl.c: preprocl.l
lex $(LFLAGS) preprocl.l
cat lex.yy.c | sed -e 's/yy/xx/g' > preprocl.c
rm lex.yy.c
lpr:
pr $(Sources) | lpr

license.o: license.c license.h

```

FIGURE 47 Uninstrumented UNIX Make File

The changes needed have been made in the modified make file shown below. The modifications are shown in bold face.

```

#####
###
##
## S A M P L E   M A K E   F I L E
##
## Make file sample, with TCAT /C instrumentation
##
## UNIX, XENIX
##
#####
###
# Uses make's knowledge of lex, yacc, cc.
#####
###

CCextras =
CFLAGS = -s ${CCextras} -DXENIX
YFLAGS = -d
LDFLAGS = -i -ly -ll
LFLAGS = -v
Lextras =
Objects = sample.o sample.y.o sample.l.o tree.o init.o error.o
dotest.o log.o \
ui.o premain.o preprocy.o preprocl.o pretree.o help.o license.o
Sources = sample.c sample.y.c sample.l.c tree.c init.c error.c
dotest.c log.c \
ui.c premain.c preprocy.c preprocl.c pretree.c sample.h type-
def.h error.h \
y.tab.h preproc.h help.c license.c license.h
# UNIX version. Compiles and links.

.c.o:
cc -P $(CFLAGS) $*.c
ic $*.i
cc -c $(CFLAGS) $*.i.c.
mv $*.i.o $*.o

```

```

#
sample: $(Objects) crun1.o
rm -f sample
cc $(Objects) crun1.o $(LDFLAGS) $(Lextras) -o sample
#
sampley.c: sampley.y
yacc $(YFLAGS) sampley.y
mv y.tab.c sampley.c
cp y.tab.h ytab.h
#
samplel1.c: samplel1.l
lex $(LFLAGS) samplel1.l
mv lex.yy.c samplel1.c
#
preprocy.c: preprocy.y
yacc $(YFLAGS) preprocy.y
cat y.tab.c | sed -e 's/yy/xx/g' > preprocy.c
cat y.tab.h | sed -e 's/yy/xx/g' > pretab.h
rm y.tab.c
#
preprocl.c: preprocl.l
lex $(LFLAGS) preprocl.l
cat lex.yy.c | sed -e 's/yy/xx/g' > preprocl.c
rm lex.yy.c
lpr:
pr $(Sources) | lpr

license.o: license.c license.h

```

FIGURE 48 Instrumented UNIX Make File

4.5.4 Running Your Make File

Now you are ready to run your program. Please follow these steps:

1. Invoke *TCAT* as you normally would (see Section 5.2).
2. Invoke the **Execute** window (see Section 5.3).
3. Make sure the **Make** command, which invokes the **make** utility, is set to the command you need. The default is set to **make**. To change it, position the mouse pointer in the specification region and then click the mouse button. When the cursor appear, edit the region accordingly.
4. You need to specify a make file name for the **Makefile name** option. The **make** utility will use this file. Simply position the mouse pointer so it is in the specification region and then click on the mouse button. When the cursor appears, type in the name of your make file.

5. Click on the **Action** pull-down menu.
6. Select the **Make** option.
7. This option will invoke the **make** utility, which will use the make file. The preprocessing, instrumenting, compiling, and linking instructions are executed.
8. The mouse pointer turns into a wristwatch symbol and the **Execute** window's options gray out until the make file's statements are executed.
9. Run your make file like any other instrumented and compiled program (see Section 5.4 for program running instructions).

4.6 Obtaining Coverage Reports

When you ran your program, all branch/call-pair coverage information was written to a trace file, default *Trace.trc* or the trace file name you specified.

To obtain coverage reports, the specified trace file is inputted into the coverage analyzer. The coverage analyzer generates coverage reports and an archive file (named *Archive*), which can be used in the second run of the coverage analyzer. The archive file is similar to trace files in its format and content. The significant difference is that the archive file does not contain information on the sequence in which logical branches were hit. It does, however, contain all other data required for coverage analysis.

The archive file is useful if you run several subsequent test sessions and want cumulative results. In such a case, both the archive file and the trace file are inputted into the coverage analyzer. This is done automatically for you.

Following is a diagram of how coverage reports are created.

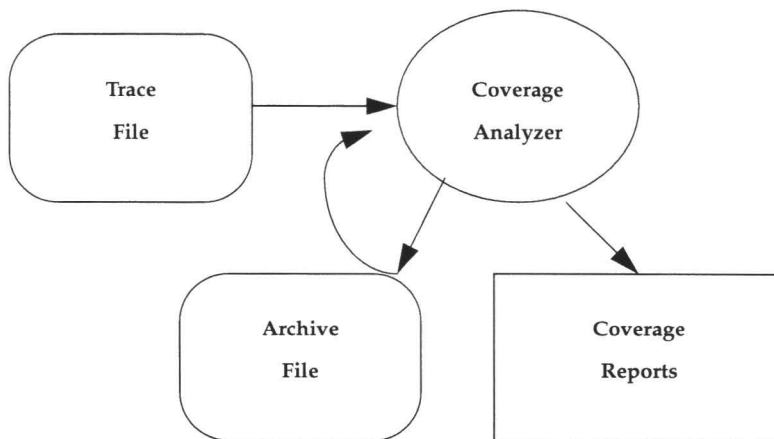


FIGURE 49 Obtaining Coverage Reports

4.6.1 Invoking the Analyze Window

Invoke the **Analyze** window from the **Main** window. Simply click on the **Analyze** button. The window below pops up.

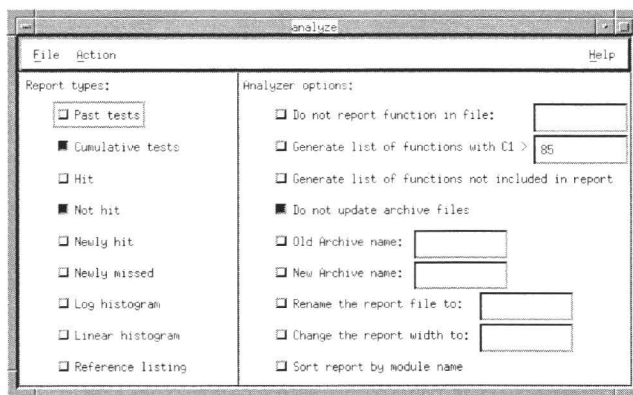


FIGURE 50 Invoking the Analyze Window

4.6.2 Selecting the Trace File Name

You must select the trace file you named when you ran the program. Eventually the trace file will be fed into the coverage analyzer to create reports. To select a file name:

1. Click on the **File** pull-down menu.
2. Select **Set Input Trace File Name**.
3. A file selection dialog box like the one below pops up.
4. Select an existing trace file name.
5. Select a file name by clicking on it in the **Files** selection window or typing it in the **Selection** entry box.

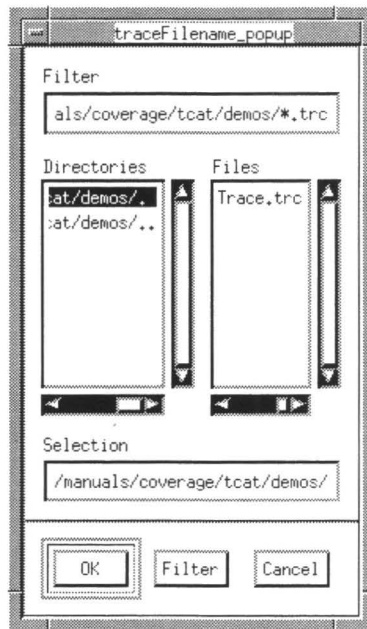


FIGURE 51 Selecting the Trace File Name

4.6.3 Selecting Reports

Before you run the trace file through the coverage analyzer, you must specify which reports you would like to see. The coverage analyzer will only take the information you specify it take from the trace file (and the archive file). For a detailed description of the below reports, please refer to the Chapter 5.

You can select any of the following check buttons by clicking once in the corresponding box. A check button is turned on if it is darkened. If it is hollowed, then it is turned off.

- **Past tests** button. The coverage analyze will produce a Past report. The Past Test report gives analysis of the archive file only. It summarizes the percentage of logical branches hit in each module, giving the C1 value for each module and the program as whole. This button is defaulted off.
- **Cumulative tests** button. The coverage analyze will produce a Cumulative report. This report tells you how many times each module was invoked, how many of its logical branches were hit, and its resulting C1 coverage measure. It analyzes information from both the trace file and the archive file. This button is defaulted on.
- **Hit** button. The coverage analyze will produce a **Hit** report. The **Hit** report identifies all of the logical branches within each modules that were exercised during your test suites. This button is defaulted off.
- **Not hit** button. The coverage analyze will produce a **Not Hit** report. The Not Hit report gives each module name and an identification number for each segment not hit in the current test. To identify the actual code not executed, look up the segment identification number in the Reference Listing report. This button is defaulted on.
- **Newly hit** button. The coverage analyze will produce a **Newly Hit** report. This report identifies which logical branches are hit in the present test which were not hit in any prior test. This button is defaulted off.
- **Newly missed** button. The coverage analyze will produce a **Newly Missed** report. This report shows which logical branches were not hit in the current execution that were hit previously. The button is defaulted off.
- **Log histogram** button. The coverage analyze will produce a **Logarithmic Histogram** report. This report demonstrates the frequency distribution of branches exercised in each module. This button is defaulted off.
- **Linear histogram** button. The coverage analyze will produce a **Linear Histogram** report. This report graphs a mark for each branch hit during testing. This button is defaulted off.
- **Reference listing** button. The coverage analyze will produce a **Reference Listing** report. This report shows the coverage level achieved for all modules that are named in the specified reference

listing, *basename.i.A*. The button is defaulted off. To obtain the **Reference Listing** report, you must specify the **Reference Listing** file. The coverage analyzer takes the information from the **Reference Listing** file and then creates a report.

Remember, the **Reference Listing File** is a version of your C program which has logical branches marked. The **Reference Listing** report has the same information, except it identifies the coverage for each module, the number of times each logical branch was hit and which were not hit. Here's how to specify the **Reference Listing** file:

1. Click on the **Reference Listing** radio button like you normally would.
2. A file selection box like the one below pops up.
3. Select an existing reference listing file, *basename. i.A*.
4. Select a file name by clicking on an already existing file in the **Files** selection window or typing in the file name in the **Selection** entry box.

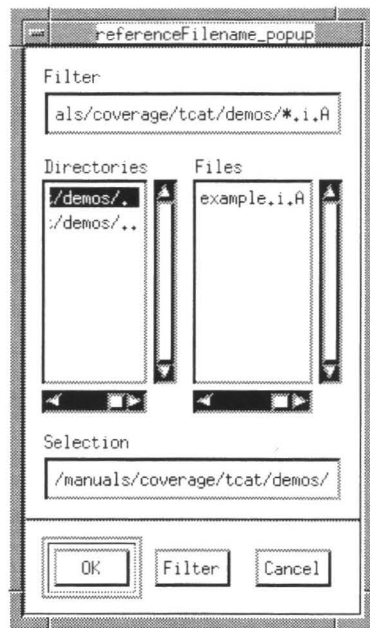


FIGURE 52 Reference Listing File Selection

4.6.4 Selecting Coverage Analyzer Options

Before you run the coverage analyzer you may want to select some of the following options, which can effect the coverage reports in various ways. For a detailed description of the below options, please refer to Chapter 5.5.

You can select any of the following check buttons by clicking once in the corresponding box. A check button is turned on if it is darkened. If it is hollowed, then it is turned off. If an option has a corresponding specification region, simply position the mouse pointer in the specification region and then click the mouse button. A cursor should appear and you can edit accordingly.

- **Do not report function in file button.** Use this option if you don't want the coverage analyzer to create coverage reports based on certain modules. You must already have a de-instrument file, defaulted to *DEINSTRU.fns* with the module(s) listed. Simply type the file name in the specification region. This option is defaulted off.
- **Generate list of functions with C1> button.** Use this option to specify a threshold value. Any module with percentage coverage greater than or equal to the threshold value (defaulted to 85) percentage will automatically be written to the de-instrument file, *DEINSTRU.fns*. Type in the threshold number in the specification region. This option is defaulted off.
- **Generate list of functions not included in report button.** Use this option to see which modules are excluded from coverage reporting. The list of excluded modules is printed at the end of the coverage report. This option should be used with the **Do not report function in file** option. This option is defaulted off.
- **Do not update archive file button.** Use this option to suppress updating the archive file. This is useful if you want the archive file to be the basis for past test information. This option is defaulted on.
- **Old Archive name button.** Use this option to include data from an old archive file in your reports. Type in the name of the old archive file in the specification region. This option is defaulted off.
- **New Archive name button.** Each time you run the Coverage Analyzer, you will write over the contents of the archive file. If you want to keep a coverage run's archive file results, you can use this option. Simply type in a different file in the specification region. If you don't include a file name, the accumulated test data

will automatically defaulted to the file name *Archive*. This option is defaulted off.

- **Rename the report file to button.** When you run the Coverage Analyzer, coverage reports are automatically written to a file named *Coverage*. If you want a different report file, use this option. Simply type in the new file name in the specification region. This option is defaulted off.
- **Change the report width to button.** Normally the reports generated by the coverage analyzer 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 option to generate reports that are wide enough to accommodate the full 128 characters. Simply type in the width in characters in the specification region. This option is defaulted off.
- **Sort report by module name button.** Use this option to produce output reports with module names sorted alphabetically. This option is defaulted off.

4.6.5 Running the Coverage Analyzer

After selecting the kinds of reports and any coverage analyzer options, you just need to run the coverage analyzer to obtain coverage reports. Here's how:

1. Click on the **Action** pull-down menu.
2. Select **Run Coverage Analyzer**.
3. The mouse pointer turns into a wristwatch symbol and the Analyze window's options gray out. During this time-out period the coverage analyzer is taking information from the trace file (and the archive file) and then creating a file named *Coverage*, which contains the coverage reports you selected.
4. When the mouse pointer is returns, the coverage analyzer has completed creating coverage reports.

4.6.6 Looking at Coverage Reports

To look at coverage reports:

1. Click on the **Action** pull-down menu.
2. Select **View Report**.
3. A **View Report** window like the one below pops up.
4. It lists which reports you selected and each subsequent report follows.

5. To look at reports, use the scroll bars to move up /down or side / side.
6. When you are finished looking at the reports, you can close the **View Report** window by clicking on the **Action** pull-down menu and selecting **Exit**.

From the coverage reports you selected, you should be able to determine which segments were exercised. We recommend that you try to obtain 85 percent coverage. If you report coverage is less than 85 percent, we recommended re-exercising your test suite. From the coverage information, you should be able to determine which segments need to be exercised.

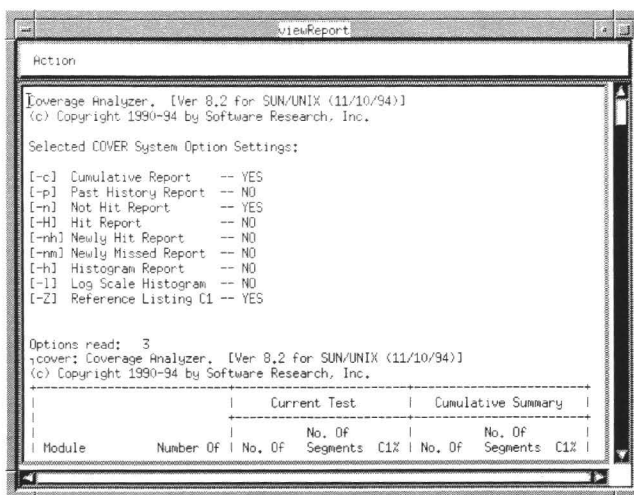


FIGURE 53 Looking at Coverage Reports

4.6.7 Exiting the Analyze Window

Before exiting the **Analyze** window, please note that you can also look at the coverage information in a graphical display (see the accompanying documentation on the **Xdigraph** utility), which can be quite useful in identifying unexercised segments.

The **Exit** option allows you to close the **Analyze** window. Here's how:

1. Click on the **File** pull-down menu.
2. Select **Exit**. The **Analyze** window closes.

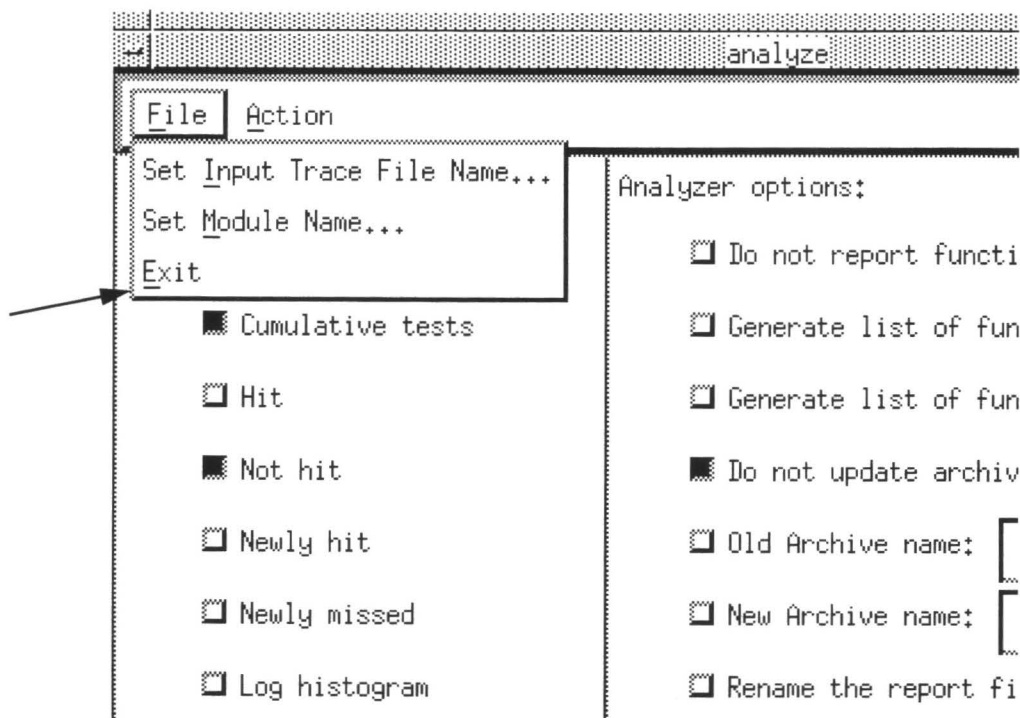


FIGURE 54 Exiting the Analyze Window

GUI Reference

This chapter defines and explains the contents of the major X Window system windows that make up the *TCAT* product, and is intended to act as a reference.

LEVEL: All users.

5.1 TCAT Menus

Once you have invoked *TCAT*, operations are initiated by using the following menus:

- **Main** window to initiate other windows.
- **Instrument** window to preprocess and instrument source programs.
- **Execute** window to compile the source program, link the program's object code to the *TCAT* object modules, and run the application.
- **Analyze** window to generate coverage reports and to analyze the control of program through graphical displays.

This chapter briefly describes the functions for each of these menus and their commands. Information on how to use these menus and commands can be located throughout Chapter 6.

5.2 Main Window

When *TCAT* is first invoked, the **Main** window is the place from which you activate other windows.



FIGURE 55 Main Window

The window has two pull-down menus (located in the menu bar):

System The **System** pull-down menu allows you to exit *TCAT*.

Help The **Help** button describes the basic functions of *TCAT*.

The window has three push buttons:

Instrument This button activates the **Instrument** window, which allows you to preprocess and instrument source programs.

Execute This button activates the **Execute** window, which allows you to compile the instrumented source code, link the instrumented program's object code with the *TCAT*-supplied runtime object modules, and run the application.

Analyze This button activates the **Analyze** window, which allows you to analyze the thoroughness of your test

suite through coverage reports and to look at graphical displays of the program's control flow.

The two pull-down menus are described on the following pages and the push buttons are described in the sections that follow.

5.2.1 System Pull-Down Menu

The **Exit** option allows you to exit *TCAT*.

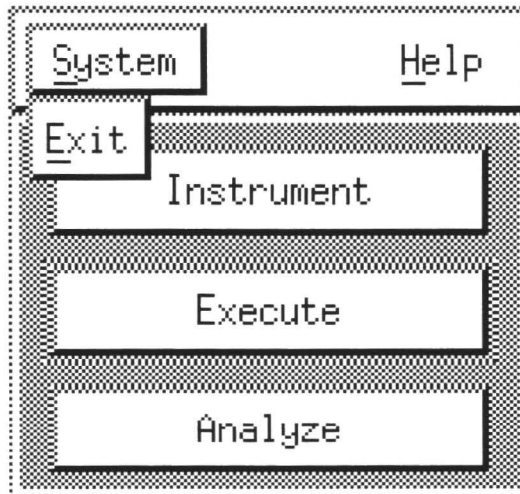


FIGURE 56 System Pull-Down Menu

5.2.2 Help Button

The **Help** button provides you with a dialog box that explains the basic operation of *TCAT*.



FIGURE 57 Help Window for the Main Window

5.3 Instrument Window

All functions necessary to preprocess and instrument source program are accessible from this window. See Section 4.3 for use of this window.

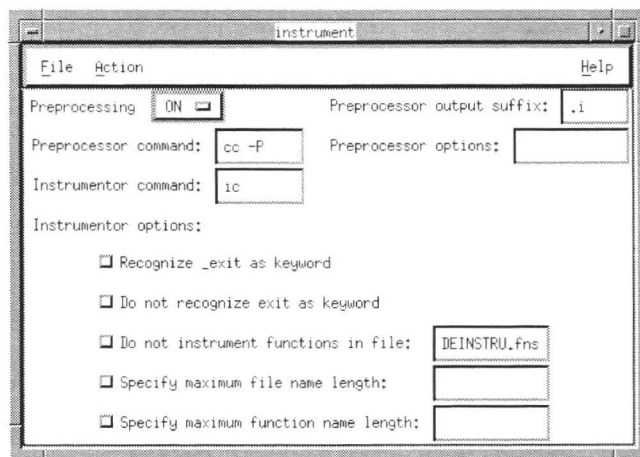


FIGURE 58

Instrument Window

The window has three pull-down menus:

- **File** pull-down menu. You use it to select the source program, or application name, and to exit the window.
- **Action** pull-down menu. You use it to preprocess and to instrument the source application.
- **Help** button. This button provides you with an on-line help menu for the **Instrument** window.

The window has an option menu:

- **Preprocessing** allows you to turn preprocessing on or off.

The window has the following specification regions.

- **Preprocessor output suffix** specification region allows you to set the suffix for the output file created from preprocessing.
- **Preprocessor options** allows you to set additional compiler options for preprocessing.
- **Preprocessor command** specification region allows you set the preprocessor command.

- **Instrumentor command** allows you to set the command that instruments the source application.
- **Instrumentor options** allows you to select a variety of options, which effect instrumentation's outcome.

Each of these options is described in the sections that follow.

5.3.1 File Pull-Down Menu

Selecting the **Set File Name** option opens up a file selection dialog box. There you can select an application name you would like to preprocess and then instrument.

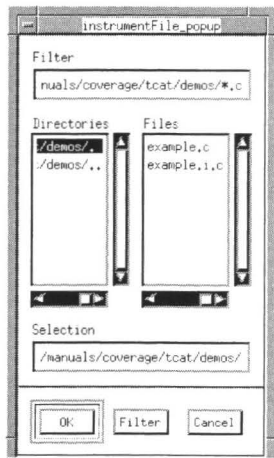


FIGURE 59 Set File Name Dialog Box
The **Exit** option closes the **Instrument** window.

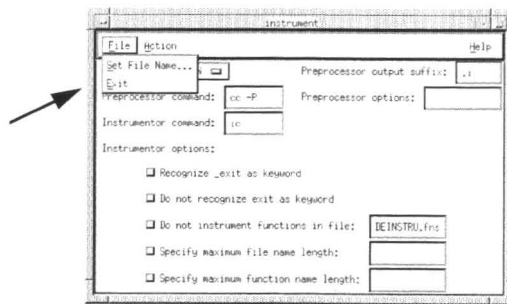


FIGURE 60 File Pull-Down Menu

5.3.2 Action Pull-Down Menu

Preprocess to preprocess the source program. Preprocessing checks your code for syntax errors prior to instrumentation.

Instrument to instrument the source program. After preprocessing, you instrument the source application. During this process, *TCAT* will automatically insert function calls at each logical branch.

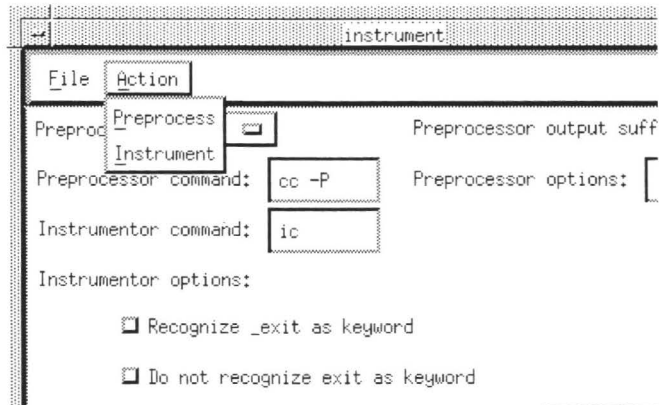


FIGURE 61 Action Pull-Down Menu

5.3.3 Help Button

The **Help** button provides you with on-line help for the **Instrument** window.

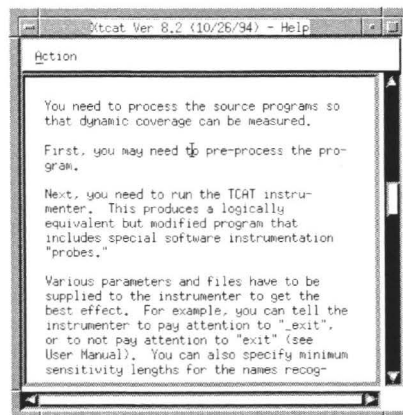


FIGURE 62 Help Window for the Instrument Window

5.3.4 Preprocessing Option Menu

The **Preprocessing** option menu allows you to turn preprocessing on or off. In most cases, you must check your program for syntax errors. In such cases, you will select the **ON** menu item.

There are times, however, you may already know there are no syntax errors and wish to forsake the preprocessing step. In these cases, select the **OFF** menu item. The default is set to **ON**.

5.3.5 Preprocessor output suffix Specification Region

After preprocessing a source program, the results are automatically written to a file. The **Preprocessor output suffix** specification region defines the suffix to that file where preprocessing results are written. The suffix is defaulted to `.i`.

5.3.6 Preprocessor command Specification Region

The **Preprocessor command** defines the command your compiler will use to compile the source program. The default is set to the standard UNIX compiler command `cc-P`.

5.3.7 Preprocessor options Specification Region

The **Preprocessor options** specification region allows you to add any additional compiler options you may want. No options are specified for the default.

5.3.8 Instrumentor command Specification Region

The **Instrumentor command** defines the command that instruments the source program. The default is set to `ic`, which is the *TCAT* standard instrumentor command.

5.3.9 Instrumentor options

If you select any of the Instrumentor options buttons, instrumentation on your source program will be effected. Below is a list of these options:

Recognize `_exit` as keyword Button

The **Recognize `_exit` as keyword** check button causes the instrumentor (`ic`) to acknowledge `exit` as a keyword. This option is defaulted off.

Do not recognize `_exit` as keyword Button

The **Do not recognize `_exit` as keyword** check button causes the instrumentor (**ic**) not to acknowledge `exit` as a keyword. This option is defaulted off.

Do not instrument functions in file Button

The **Do not instrument functions in file** check button causes the instrumentor to selectively de-instrument named functions in the file specified in the specification region. This file name is defaulted to *DEINSTRU.fns*. During the instrumentation process, the instrumentor will not mark the segments for modules named in the file. This option is recommended if you know certain modules have already been thoroughly exercised.

This option is defaulted off.

Specify maximum file name length Button

After instrumentation, the instrumentor creates the following files:

- *basename.i.c* -- an instrumented version of your "C" program, *basename*.
- *basename.i.A* -- a Reference Listing, which has the logical branches marked as Segment 1, Segment 2....
- *basename.i.S* -- an Instrumented Statistics file, where various kinds of statistics are listed for each module, including the number of statements, segments, conditional statements, etc.
- *basename.i.L* -- a Segment Count Listing file, which contains a complete count of all the modules and their segments in the program being tested.
- *modulename.dig* -- a Directed Graph Listing file for each module, which reports the segment relationship between nodes. You can also visually look at a module's directed graph using *the TCAT Xdigraph* utility (see the accompanying Software Product Notes)
- *basename.i.E* -- a Error Listing file, which contains all the errors found during instrumentation.

The **Specify maximum file name length** check button causes the instrumentor to put a limit on the amount of characters a *basename* file name can have. If the length exceeds the value specified in the specification region, then the instrumentor output files will be redirected to files named *Temp.i.?*.

The default is turned off.

Use this option when your system has a limit on the length of file name characters.

Specify maximum function name length Button

The **Specify maximum function name length** check button causes the instrumentor to put a limit on the amount of characters a function name can have. If the length exceeds the value specified in the specification region, then the instrumentor will only recognize as distinct only the first value characters of the function name. If you specify 5, for instance, then only up to the first five letters of the function names are recognized.

The default is turned off.

5.4 Execute Window

All functions necessary to compile source programs, link the programs' object modules with *the* TCAT runtime object modules and run applications are accessible from this window. See Section 5.5 for use of this window.

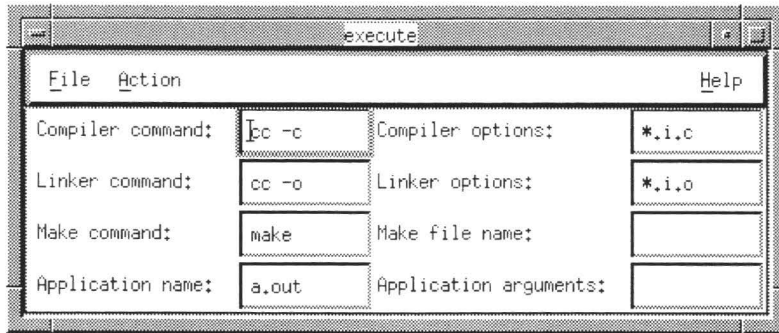


FIGURE 63 Execute Window

The window has three pull-down menus:

- **File** pull-down menu. You use it to select the TCAT runtime object module and to exit the window.
- **Action** pull-down menu. You use it to compile the instrumented program, link object files, and run the application.
- **Help** button. This button provides you with an on-help of the Execute window.

The window has the following specification regions:

- **Compiler command** specification region allows you to set the command to compile your instrumented program.
- **Compiler options** allows you to specify the instrumented programs to be compiled.
- **Linker Command** specification region allows you set the command to link the instrumented program's object code with the TCAT object modules.
- **Linker options** allows you to specify the object code file needed for linking.
- **Make command** allows you to select the command that will invoke the make utility.
- **Make file name** allows you to specify the make file.

- **Application name** allows you to name the instrumented executable.
- **Application argument** allows you to specify arguments or switches for the application.

Each of these options is described in the sections that follow.

5.4.1 File Pull-Down Menu

Selecting the **Set Runtime Obj. Module** option opens up a file selection dialog box. There you can select a *TCAT* runtime object module. The runtime object module you select is eventually linked with the instrumented program's object code, creating an executable for the application. Each runtime routine can change the performance of the instrumented system.

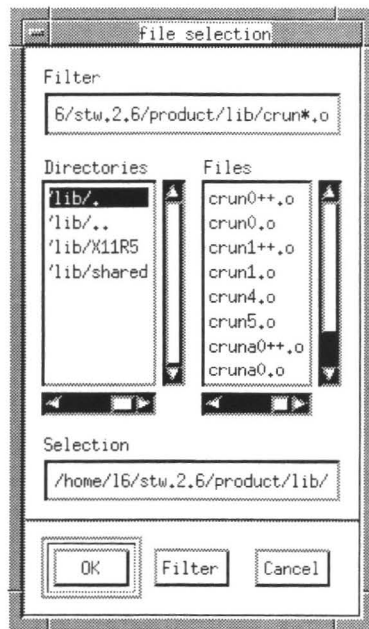


FIGURE 64 Set Runtime Obj. Module Selection Dialog Box
The Exit option closes the **Execute** window.

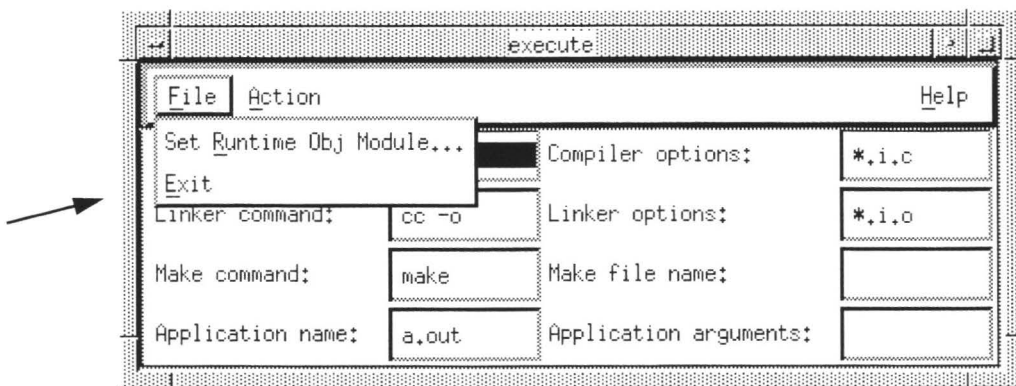


FIGURE 65 File Pull-Down Menu

5.4.2 Action Pull-Down Menu

Compile to check your instrumented program for syntax errors. When errors are found, they are displayed in the invocation window. Compiling also automatically creates an object file, *basename.i.o*, which contains object code information. This file is later linked with one of the *TCAT* object modules.

Link to link the program's object code to one of the *TCAT* object modules.

Make will invoke the make utility, which will use the make file you specify in the the **Make file name** option. This file should contain instructions to preprocess, instrument, compile and link. Creating a make file can save you a lot of time. Please refer to Section 5.4.9 for further information.

Run application to execute the program. When you run a program, you will be using your test suite to exercise module's segments as thoroughly as possible.

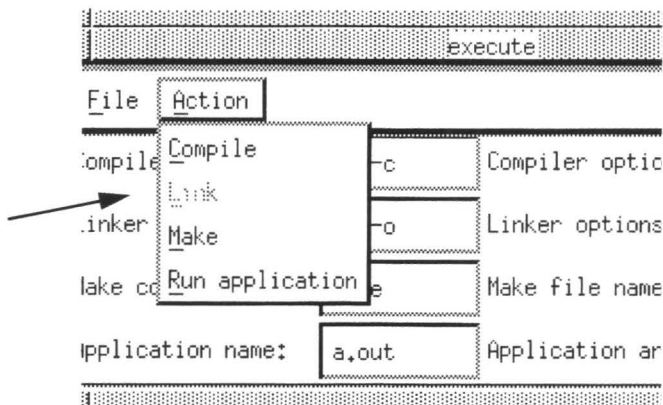


FIGURE 66 Action Pull-Down Menu

5.4.3 Help Button

The **Help** button provides you with on-line help for the **Execute** window.

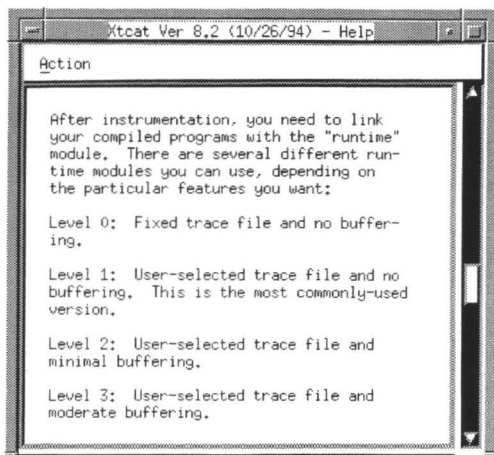


FIGURE 67 Help Window for the Execute Window

5.4.4 Compiler command Specification Region

The **Compiler command** specification region specifies the command to compile. The default is set to `cc -c`, which is the standard UNIX compiling command.

5.4.5 Compiler options Specification Region

You must specify the instrumented files to be compiled. The **Compiler options** specification region allows you to specify the instrumented programs' suffix. Generally when programs are instrumented, the instrumented version of the source program is generally named `basename.i.i.c`, unless otherwise specified in the Instrument window's Preprocessor output suffix option. For this reason, the default is set to `*i.c`.

5.4.6 Linker Command Specification Region

The **Linker Command** specification region specifies the command to link object files: a supplied SR runtime object module with object code files, `basename.i.o`. The default is set to `cc -o`.

5.4.7 Linker options Specification Region

You must specify the input object files to be linked with one of SR's object modules. These input object files are created during compiling, generally named `basename.i.o`. For this reason, the default for the **Linker options** specification region is set to `*i.o`.

5.4.8 Make command Specification Region

The **Make command** specification region allows you to specify the command used for invoking the make utility. The make utility performs the instructions defined in a make file. The default is set to `make`.

NOTE: This option is only necessary if you are using make files.

5.4.9 Make file name Specification Region

The **Make file name** specification region allows you to specify the file names for the make file. There is no default set.

NOTE: This option is only necessary if you are using make files.

5.4.10 Application name Specification Region

When object files are linked, an executable is created. The **Application name** specification region allows you specify the instrumented executable name. The default is set to *a.out*.

5.4.11 Application argument

The **Application argument** specification region allows you to add switches for the application. There is no default set.

5.5 Analyze Window

All functions necessary to look at coverage reports and to view a module's directed graph are accessible from this window. See Section 4.6.1 for use of this window.

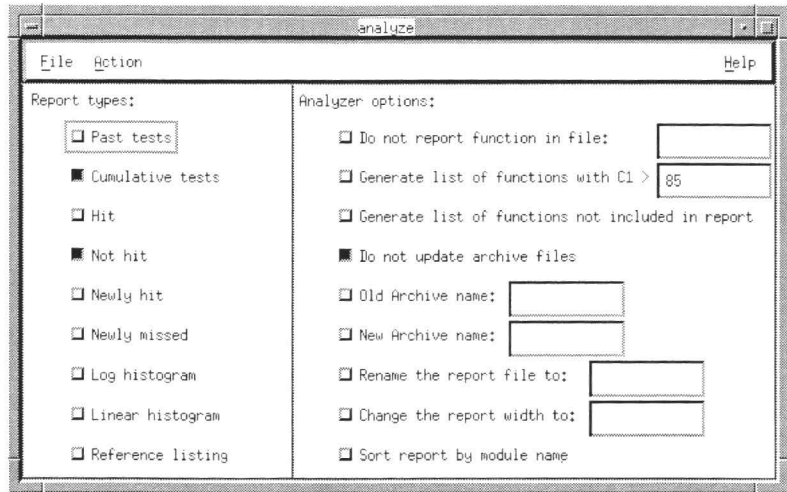


FIGURE 68

Analyze Window

The window has three pull-down menus:

- **File** pull-down menu. You use it to select the trace file and to exit the window.
- **Action** pull-down menu. You use it to run the coverage analyzer, view reports, and view source code for a program module's segments.
- **Help** button. This button provides you with an on-help of the Analyze window.

The window has the following reports available:

- **Past tests** check button.
- **Cumulative tests** check button.
- **Hit** check button.
- **Not Hit** check button.
- **Newly hit** check button.
- **Newly missed** check button.

- **Log histogram** check button.
- **Linear histogram** check button.
- **Reference Listing** check button.

The window has the following coverage analyzer options available:

- **Do not report function in file** check button.
- **Generate list of functions with C1>** check button.
- **Generate list of functions not included in report** check button.
- **Do not update archive file** check button.
- **Old Archive name** check button.
- **New Archive name** check button.
- **Rename the report file to:** check button.
- **Change the report width to:** check button.
- **Sort report by module name** check button.

Each of these options is described in the sections that follow.

5.5.1 File Pull-Down Menu

Selecting the **Set Input Trace File Name** option opens up a file selection dialog box. There you can select the trace file you named when you ran the program. Remember that when a program is run, all segment information is written to the trace file.

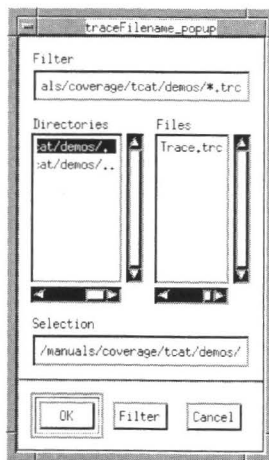


FIGURE 69 Set Input Trace File Name Selection Dialog Box
The **Exit** option closes the **Analyze** window.

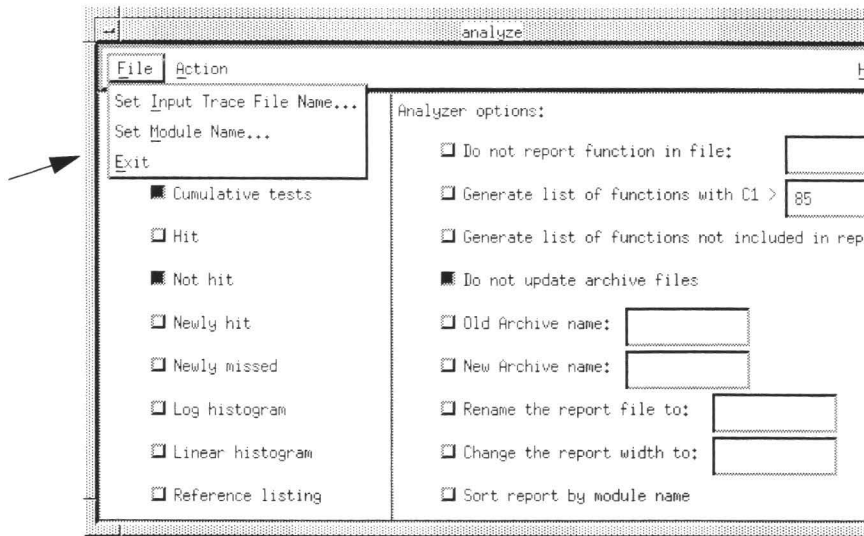


FIGURE 70 File Pull-Down Menu

5.5.2 Action Pull-Down Menu

When you use *TCAT*, you first select the trace file, then the types of reports you want and any coverage analyzer options. The **Run Coverage Analyzer** reads in the information from these three sources. The coverage analyzer then creates a file named *Coverage*, which contains the coverage reports you selected.

The **View Source** option allows you select a module from the program, look at its directed graph, and view source code for a particular segment. Please see the accompanying documentation on the **Xdigraph** utility for usage.

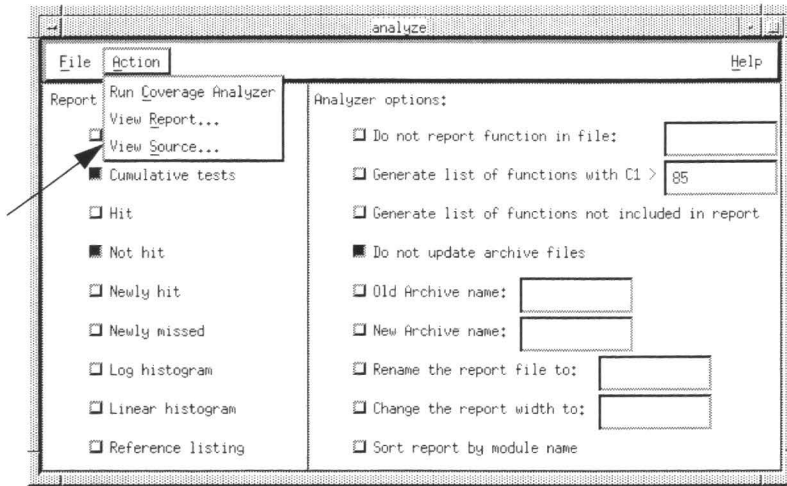


FIGURE 71 Action Pull-Down Menu

5.5.3 Help Button

The **Help** button provides you with on-line help for the **Analyze** window.

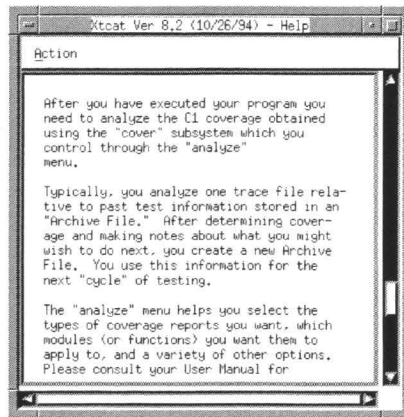


FIGURE 72 Help Window for the Analyze Window

5.5.4 Past tests Check Button

The **Past tests** button tells the coverage analyzer to produce a **Past** report. The **Past Test** report gives analysis of the Archive file only. It summarizes the percentage of segments hit in each module, giving the C1 value for each module and the program as whole. This button is defaulted off.

viewReport

Action

(c) Copyright 1990-94 by Software Research, Inc.

(Archived) Past Tests				
Module No. Name	Number Of Segments	Number Of Invocations	Number Of Segments Hit	Percent Coverage
0: example_main	27	1	21	77.78
1: example_proc_input	24	15	15	62.50
2: example_chk_char	3	15	2	66.67
Totals	54	31	38	70.37

Current test message(s) (saved in archive):
 quick start test
 ycover: Coverage Analyzer, [Ver 8.2 for SUN/UNIX (11/10/94)]
 (c) Copyright 1990-94 by Software Research, Inc.

Current Test				Cumulative Summary			
Module	Number Of	No. Of	No. Of	C1%	No. Of	No. Of	C1%
		No. Of	Segments		No. Of	Segments	

FIGURE 73 Past Report

5.5.5 Cumulative tests Check Button

The **Cumulative test** button tells the coverage analyzer to produce a **Cumulative** report. This report tells you how many times each module was invoked, how many of its segments were hit, and its resulting C1 coverage measure. It analyzes information from both the trace file and the Archive file.

This button is defaulted on.

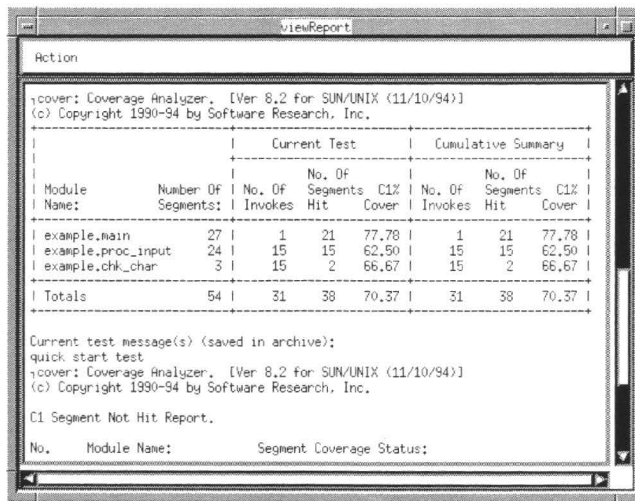
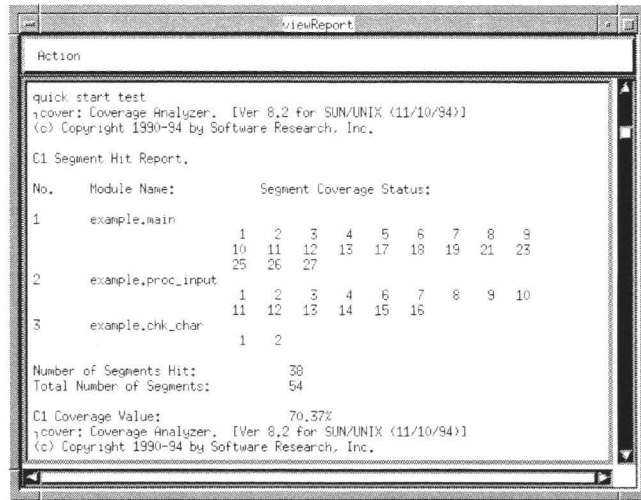


FIGURE 74 Cumulative Report

5.5.6 Hit Check Button

The **Hit** button tells the coverage analyzer to produce a **Hit** report. The **Hit** report identifies all of the segments within each modules that were exercised during your test suites. This button is defaulted off.



```

Action
-----
quick start test
ycover: Coverage Analyzer. [Ver 8.2 for SUN/UNIX (11/10/94)]
(c) Copyright 1990-94 by Software Research, Inc.

C1 Segment Hit Report.

No.   Module Name:           Segment Coverage Status:
-----
1     example.main
      1  2  3  4  5  6  7  8  9
      10 11 12 13 17 18 19 21 23
      25 26 27
2     example.proc_input
      1  2  3  4  6  7  8  9  10
      11 12 13 14 15 16
3     example.chk_char
      1  2

Number of Segments Hit:      38
Total Number of Segments:   54

C1 Coverage Value:          70.37%
ycover: Coverage Analyzer. [Ver 8.2 for SUN/UNIX (11/10/94)]
(c) Copyright 1990-94 by Software Research, Inc.
  
```

FIGURE 75 Hit Report

5.5.7 Not Hit Check Button

The **Not Hit** button tells the coverage analyzer to produce a **Not Hit** report. The **Not Hit** report gives each module name and an identification number for each segment not hit in the current test. To identify the actual code not executed, look up the segment identification number in the **Reference Listing** report. This button is defaulted on.

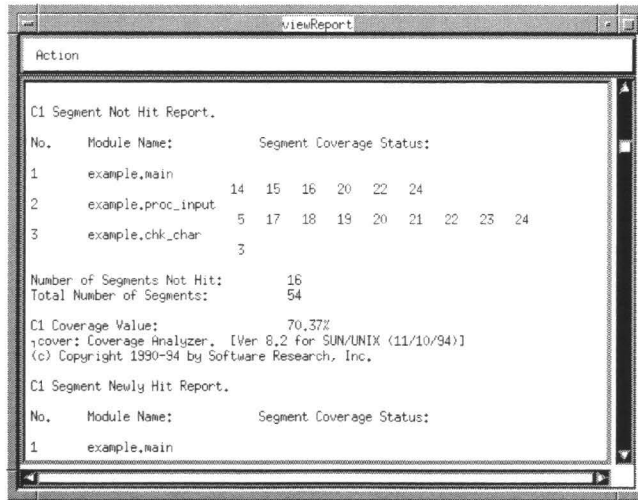


FIGURE 76 Not Hit Report

5.5.8 Newly Hit Check Button

The **Newly Hit** button tells the coverage analyzer to produce a **Newly Hit** report. This report identifies which segment are hit in the present test which were not hit in any prior test.

This button is defaulted off.

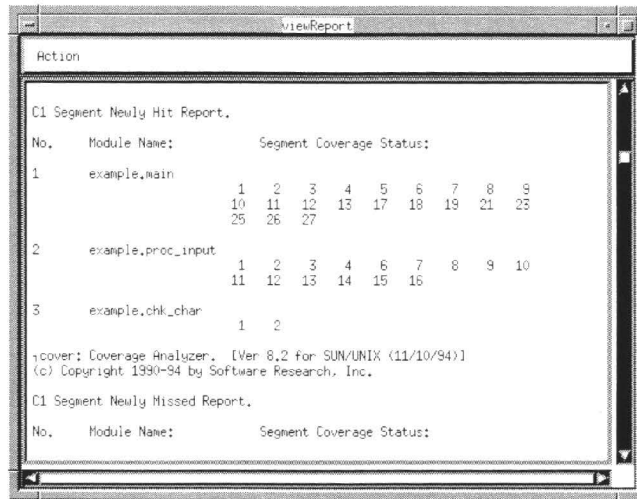


FIGURE 77 Newly Hit Report

5.5.9 Newly missed Check Button

The **Newly missed** button tells the coverage analyzer to produce a **Newly Missed** report. This report shows which segments were not hit in the current execution that were hit previously.

The button is defaulted off.

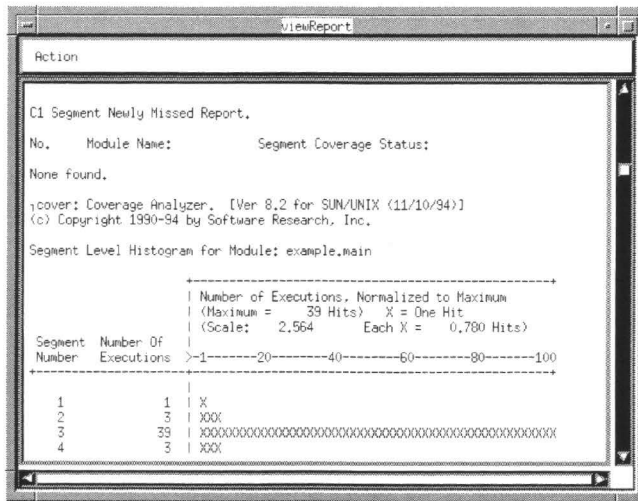


FIGURE 78 Newly Missed Report

5.5.11 Linear histogram Check Button

Linear Histogram button tells the coverage analyzer to produce a **Linear Histogram** report. This report graphs a mark for each branch hit during testing. This button is defaulted off.

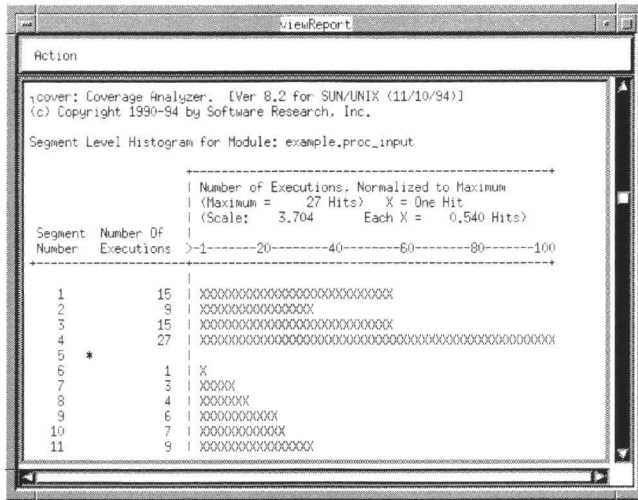


FIGURE 80 Linear Histogram Report

5.5.12 Reference listing Check Button

The **Reference listing** button tells the coverage analyzer to produce a **Reference Listing** report. This report shows the coverage level achieved for all modules that are named in the specified reference listing, *basename.i.A*. If a module is tested but the name is not found in the supplied reference listing file, then the that coverage is not reported. Similarly, if a name appears in the reference listing but is not found in the trace file or the Archive file, no coverage will be reports.

The button is defaulted off.

To obtain the **Reference Listing** report, you must specify the **Reference Listing** file from a file selection dialog box (shown below). This file is a version of your "C" program which has logical branches marked. The coverage analyzer takes the information from the **Reference Listing** file and then creates a report.

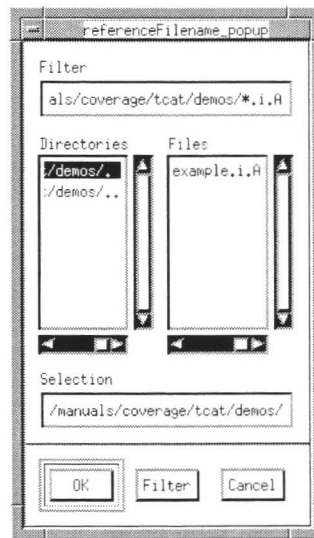


FIGURE 81 Reference Listing File Selection


```

Action
-----
-- TCAT/C. Ver 8.2 for SUN (10/28/94).
--
-- (c) Copyright 1990 by Software Research, Inc.  ALL RIGHTS RES
-- SEGMENT REFERENCE LISTING
-- Instrumentation date: Tue Jun 20 08:27:55 1995
-- Separate modules and segment definitions for each module are
-- indicated in this commented version of the supplied source fi
-----
extern struct _iobuf {
    int    _cnt;
    unsigned char *_ptr;
    unsigned char *_base;
    int    _bufsiz;
    short  _flag;
    char   _file;
} _iobuf[];
extern struct _iobuf  #fopen();
extern struct _iobuf  #fdopen();
extern struct _iobuf  #freopen();

```

FIGURE 82 Reference Listing Report

5.5.13 Do not report function in file Check Button

The **Do not report function in file** button specifies the de-instrumentation file. This file lists the modules you do not want the coverage reports to reflect. The default file is set to *DEINSTRU.fns*.

This option is defaulted off.

5.5.14 Generate list of functions with C1> Check Button

The **Generate list of functions with C1>** button specifies the module threshold value. Any module with percentage coverage greater than or equal to the threshold value (defaulted to 85) percentage will automatically be written to the de-instrument file, defaulted to *DEINSTRU.fns*.

This option is defaulted off.

5.5.15 Generate list of functions not included in report Check Button

The **Generate list of functions not included in report** button tells the coverage analyzer to print the list of modules excluded from instrumentation to be printed at the end of the coverage reports.

This option is used with the **Do not report function in file** option. This option is defaulted off.

5.5.16 Do not update archive file Check Button

The **Do not update archive file** button check button suppresses the coverage analyzer from updating the Archive file for the current run. This is useful if you want the Archive file to be the basis for past test information.

This option is defaulted on.

5.5.17 Old Archive name Check Button

Use this option to include data from an old Archive file in your reports. You must specify the name of the old Archive file.

This option is defaulted off.

5.5.18 New Archive name Check Button

Each time you run the Coverage Analyzer, you will write over the contents of the Archive file. If you want to keep a coverage run's Archive file results, you can use this option. You must specify the new Archive name in the specification region. If you don't include a file name, the accumulated test data will automatically defaulted to the file name Archive.

This option is defaulted off.

5.5.19 Rename the report file to: Check Button

The **Rename the report file to:** button automatically allows you to specify a file where coverage reports are written. This file is generally named Coverage. If you want a different report file, use this option.

This option is defaulted off.

5.5.20 Change the report width to: Check Button

The **Change the report width to:** button allows you change the report width, which is defaulted to 80 characters. Normally the reports generated by the coverage analyzer are wide enough to accommodate module names up to 80 characters in length. The internal limit on name length is, however, 128 characters. You can use this option to generate reports that are wide enough to accommodate the full 128 characters.

This option is defaulted off.

5.5.21 **Sort report by module name Check Button**

The **Sort report by module name** button tells the coverage analyzer to produce output reports with module names sorted alphabetically.

This option is defaulted off.

Command-Line Activation

This chapter describes in detail the various command-line switches which perform tasks similar to the X Window System graphical user interface (GUI).

LEVEL: If you are a beginning or intermediate *TCAT* user, you can skip this section on first reading; it is intended for advanced users.

6.1 Command Line Usage

This chapter describes the main operating modes of *TCAT*. After working with the GUI, you may want to work with the command line. Many of the available command line options are equivalent to functions that can otherwise be performed by choosing commands from the *TCAT* graphical user interface. For experienced users it can mean more efficient testing.

6.2 'Xtcat' Command

You invoke the *Xtcat* system with the command:

```
xtcat [-L lang]
```

Options and Parameters:

- | | |
|----------------|--|
| No Options | Invokes <i>Xtcat</i> for the "C" language interactively. |
| -L <i>lang</i> | Specifies the language. The following languages are supported: |
| • -L C | Supports the "C" language. This is the default. |
| • -L C++ | Supports the "C++" language. |
| • -L Ada | Supports the Ada language. |
| • -L F77 | Supports the FORTAN language. |

6.3 ic Instrumentor Command

TCAT instruments the source code of the system to be tested, that is it inserts function calls at each logical branch. The instrumentation will not affect the functionality of the program. When it is compiled, linked and executed, the instrumented program will behave as it normally does, except that it will write coverage data to a trace file. There is some perfor-

mance overhead due to the data collection process. The trace file is processed by a report generator.

As already mentioned, an instrumented program is one that has been specially modified so that, when executed, it transmits information about C1 coverage at every stage of testing while behaving logically equivalent to the original program.

In its operation, *TCAT's* instrumentor parses your candidate source code, looking for logical branches. When one is discovered, the instrumentor inserts a function call in the instrumented version of the source code. It is important to note that the resulting source code file is still a legal program written in C, as was the original program. The only difference is the added function calls.

When executed, the inserted function calls write to a trace file. Remember, the instrumented version will otherwise function as the uninstrumented version.

The complete syntax for command line calls to `ic`, or the instrumentor, is listed below.

Options and Parameters:

```
ic [option[s]] basename .i
  [option[s]]
-ce
-cw
-dil-DI file
-fn number
-fl number
-h -help
-I
-lj
-m
-m6
-n
-t
-u
-w
-x
-z
```

The `ic` command instruments submitted C language files. It takes a *base-name.i* file produced from preprocessing and produces an instrumented version of the source called *basename.ic*.

It is required that you first preprocess the source file through a C preprocessor before passing it to `ic`. The preprocessing command is:

```
cc -P filename .c
```

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

- file.ext* [file.ext] File(s) to be instrumented. *ext* can be *c* or *i*. If there are multiple files, then each is processed in the order presented.
- ce** Conditional Expression Processing Switch. If this switch is present, the instrumenter will process conditional expressions of the form *? a : b* found in the submitted programs.
- cw** Conditional Expression Warning Suppression Switch. Normally, conditional expressions are not processed and the following warning is issued to the user that a conditional expression was found: "Conditional Expressions Not Processed Warning" message. (See the **-ce** switch explanation.)
- When this switch is present, the instrumenter does not warn you when a conditional expression is found, and does not process it. When this switch is not present, the instrumenter warns you instruments the logical predicate associated with a *? a : b* type expression.
- DI file** De-instrumented Switch. Allows you to specify a line of modules that are to be excluded from coverage reporting. Only the list of module names found in the specified *file* is to be excluded from coverage reporting. The module names can be specified in any format. White space (tabs, spaces) is ignored.
- f1 number** Allows you to specify the maximum length of file name characters that are allowable on the system. If the length of a generated file name exceeds the *number*, then the instrumentor output will be redirected to files named *Temp.i.?*. These files can be used in subsequent processing.
- fn number** The Flexname Switch. Allows you to specify the maximum characters of function names the instrumentor recognizes. If the function name exceeds the *number*, then the instrumentor will recognize as distinct only the first *number* characters of the function name. For instance, **-fn 5** will recognize the first five characters as distinct. Characters beyond that point, howev-

er, will not be recognized for function name purposes.

- h**
- help** Help Switch. Forces output to show a summary of available switches. Note : This is also the output produced by an illegal command.
- I** Ignore Errors Switch. In certain rare cases, when the underlying C compiler supports non-standard options and constructs, it may be desirable to “force” instrumentation to occur regardless of errors found. This is done with the **-I** switch.

CAUTION: When instrumentation is forced using this switch, there is a chance that the instrumented software will not compile. For example, if you use the **-I** switch to “instrument” a file of text material, you would not expect the output to be compilable (and it probably won’t be), even though it may have been “instrumented”.
- lj** Process Set-Jump, Long-Jump Switch. If present, processes `setjmp` and `longjmp` statements found in the submitted C programs. If this switch is not present, these statements may cause an error during instrumentation. Applies only to UNIX.
- m** Recognizes Microsoft C 5.1 keywords during the instrumentation process.
- m6** Recognize Microsoft C 6.0 keywords during the instrumentation process.
- n** No Null Edge Instrumentation Switch. Normally, the instrumentor finds empty edges and instruments them. If this switch is used, then such extra instrumentation is suppressed. This will affect the instrumentation of `if` and `switch` statements that do now have an `else` statement.
- t** Recognize Turbo C keywords during the instrumentation process.
- u** Forces the instrumentor to recognize `_exit` as a keyword.
- w** Recognize Whitesmith C keywords during the instrumentation process.
- x** Will not recognize `exit` as a keyword.

-z Recognize MANX/AZTEC C keywords during the instrumentation process.

If there is an error, **ic** gives a response line, or usage line, indicating the set of possible switches and options, which is the same as the **-h** output.

You can also look at the available options by entering **ic -help**. The following will appear on your display:

```
TCAT/C Instrumenter/Analyzer, Release 8.2 for SUN (09/14/92).
Legal options are:

[-h] Show options
[-I] Ignore errors
[-m] Recognize Microsoft 5.1 keywords
[-m6] Recognize Microsoft 6.0 keywords
[-t] Recognize Turbo C keywords
[-u] Recognize "_exit" as exit
[-w] Recognize Whitesmith C keywords
[-x] Will not recognize exit as keyword
[-z] Recognize MANX/AZTEC C keywords
[-di file] Specify the file contains list of functions not to
instrument
[-lj] Recognize setjmp/longjmp as global goto
[-cw] Do not report warning message on conditional expression
[-ce] Allow instrumentation of conditional expression
[-n] Do not instrument empty edges (ie. "else" and "default")
[-fl number] Specify the maximum file name length output will go
to Temp.i.*
[-fn number] Specify the maximum length of function names
```

6.3.1 File Summary

This section describes *TCAT* file naming conventions for the instrumentor **ic**.

```
ic [optional switches] basename.i
```

Input:

basename.i Preprocessed source file

Produces:

basename.i.c Instrumented source

basename.i.A Segment reference listing

basename.i.E Error listing

basename.i.L Segment and count (Used by the *mkarchive* utility)

basename.i.S Instrumentation Statistics

modulename.dig Module digraph file(s) (Used by **Xdigraph** source viewing utility).

Please see Section 6.5 for further information on the **mkarchive** utility and the supplied documentation on, the **Xdigraph** utility.

6.3.2 Instrumentation Directive

The *TCAT* system permits use of a special passive “directives” in the form of a comment statement that can be used to turn the instrumentation process ON or OFF within a module’s boundary. These comment statements control C1, S1, and both C1 and S1 instrumentation.

Because these comments are passive, they can safely be placed in the original source code so that successive re-instrumentations will follow the same non-interfering directives.

Application of Directive

You can use these directives to prevent instrumentation that would otherwise produce too much output of that applies to a passage that does not need to be further tested.

The de-instrumentation directive feature can, with some limitations (see the examples on the following page), let you avoid instrumenting part of a C module. To do this, bracket the passage of code with: `/*TCAT OFF */` and `/*TCAT ON */` to turn off instrumentation for C1 and S1; `/*TCAT SCAN OFF */` and `/*TCAT SCAN ON */` to bypass all of the information in the passage; `/*TCAT C1 */` and `/*TCAT C1 ON */` to turn off instrumentation for C1; and `/*TCAT S1 */` and `/*TCAT S1 ON */` to turn off instrumentation for S1.

Note that in addition to these directives, there is also an automatic de-instrumentation feature (`-di file`) that allows for selective de-instrumentation of individual C functions. With this option, you may specify the name of a function that is not to be instrumented, and the *TCAT* process will disregard that name if it finds it. This effectively ignores entire modules from the instrumentation process.

Proper Directive Placement

Basically, you have the capability to turn on and turn off entire C structures within the program. However, the directives can be placed only in certain locations within your C program, as shown next.

Processing of a file always begins with directive processing ON. There can be as many directive instances in a program as you want. However, they cannot span over a function definition boundary.

All the directives should be used in the same manner. Below are examples of how you can use these directives:

1. Between the body of function, that is between the {...} of a function, as shown below.

```
...
procedure example ()
{
body
}

procedure example ()
{
/* TCAT OFF */
body
/* TCAT ON */
}
...
```

or

```
...
procedure example ()
{
body
}

procedure example ()
{
/* TCAT SCAN OFF */
body
/* TCAT SCAN ON */
}
...
```

or

```
...
procedure example ()
{
body
}
```

```
procedure example ()
{
/* TCAT C1 OFF */
body
/* TCAT C1 ON */
}
...
```

or

```
...
procedure example ()
{
body
}

procedure example ()
{
/* TCAT C1 OFF */
body
/* TCAT C1 ON */
}
...
```

- 2. Before the first statement of an if or while or for or switch construct. In this case the placement has to be as shown in these examples:**

```
...
if (...) { body }

/* TCAT OFF */
if (...)
{ body }
/* TCAT ON */
...
.fi
.bp
```

or

```
...
if (...) { body }

/* TCAT SCAN OFF */
if (...)
```

```
    { body }  
    /* TCAT SCAN ON */  
    ...
```

or

```
    ...  
    if (...) { body }  
  
    /* TCAT C1 OFF */  
    if (...)  
    { body }  
    /* TCAT C1 ON */  
    ...
```

or

```
    ...  
    if (...) { body }  
  
    /* TCAT S1 OFF */  
    if (...)  
    { body }  
    /* TCAT S1 ON */  
    ...
```

Here is the same kind of construction for a `while` :

```
    ...  
    while (...) { body }  
  
    /* TCAT OFF */  
    while (...)  
    {  
    body  
    }  
    /* TCAT ON */  
    ...
```

or

```
    ...  
    while (...) { body }  
  
    /* TCAT SCAN OFF */  
    while (...)  
    {
```

```
body
}
/* TCAT SCAN ON */
...

or

...
while (...) { body }

/* TCAT C1 OFF */
while (...) {
body
}
/* TCAT C1 ON */
...

or

...
while (...) { body }

/* TCAT S1 OFF */
while (...) {
body
}
/* TCAT S1 ON */
...
```

Improper Directive Placement

The placement of the directives cannot cross structural boundaries, and the span from a `/* OFF */` to a `/* ON */` cannot cross a function definition.

Below is an example of an illegal construction for a `/* TCAT OFF */` \ON directive construction. This construction will result in compilation errors:

```
...
/* TCAT ON */
if (...) /* TCAT OFF */ { body
while (...) {
/* TCAT ON */
body
}
}
/* TCAT OFF */
...
```

Additional Notes

You can have as many pairs of the directives in any one file as you want. However, the directives' pairs cannot span a function definition boundary.

You can have multiple directives in any one function. In fact, you may want to disable instrumentation in the innermost loops in a function that is used a great deal as a way of keeping the instrumentation overhead low.

6.4 cover Command

To get useful results from *TCAT*, you must analyze coverage reports. To do this, the program `cover` is run to process the trace file and produce several output reports. The **cover** command analyzes trace files produced by instrumented programs and generates a set of coverage reports.

Reports generated by **cover** are stored by default in the file *Coverage*. These reports are useful for performance analysis and also for "hot spot" tuning. Depending on the options used, **cover** produces different reports.

cover also archives the trace file information into an *Archive* file so that the reports are cumulative.

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

Options and Parameters

```
cover [tracefile[s] [option[s]]
      [option[s]]
-a file
-b file
-c
-d [ name[ s ]]
-DI|-di file
-DL
-f file
-h name[ s ]]
-help
-H
-l name [ s ]]
-m
-n|-N
-NH
-nl file
-NM
-p
-q
```

```
-r file  
-s  
-SU  
-T [ # ]  
-w width  
-Z file
```

[*tracefile[s]*] is the name of the trace file(s) that you wish for the coverage analyzer to process. If there are no trace files given, then *cover* looks for data in the default trace file name, *Trace.trc*. If there are no names given and *Trace.trc* is not present, 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.

The options are listed in alphabetical order .

- a file** Old Archive File Name Switch. Allows you to 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" column test results, but not under the column "Current 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" column figures will be the same as those for "Current 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 named *file* 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.
- d name [s]** Module Name Delete Switch. Deletes named modules from the generated *Archive* file, if found in the current execution. Subsequently, *cover* will never have heard about these names. This switch is useful

? [-DI | -di file

-DL

-f file

-h name[s]

-l name[s]

in updating an extensive test record that would otherwise be lost due to the complexity of editing the Archive file.

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

De-instrumented Module List Switch. Allows you 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.

New Archive File Name Switch. Places newly accumulated test coverage data in the file you specify. If you don't include a different name with this switch, the accumulated test data will be placed in the default name Archive .

CAUTION: Each time you run `cover`, 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.

Linear Histogram Report Switch (-h).

Logarithmic Histogram Report Switch (-l).

These two options produce two histogram reports that graph the frequency distribution of the logical branches exercised in a single module. The histograms provide a module-by-module analysis of testing coverage, combining current trace file data with archive data 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, cover 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. This option produces the Hit report. It 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.
- help** Help Switch. Shows a summary of available switches.
- m** Minimal Output Switch. When present, cover 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. This option produces the Newly Hit report. Shows the segments by module that were hit in the current execution that were not hit previously. Thus this gives you 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 file** Name List Switch. This switch specifies that only the list of module names found in the specified *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 *file*:
- 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 but 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. This option produces the Past 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 `cover` 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.
- s** Sort Switch. This option produces output reports with module names sorted alphabetically.
- SU** Suppress Update Switch. During processing, `cover` will suppress updating of the archive file, either the default *Archive* or the file named by the **-f** switch. `cover` will read the data in the archive file to form the basis for the "past test" information.
- T [#]** Coverage Threshold Switch. # 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** 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 `cover` 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 `cover` 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.

`-z file`

Annotated Reference Listing Switch. `cover` will analyze the specified archive file, any specified trace files, and will produce a report that shows the coverage level achieved for all modules that are named in the specified reference listing (files with a `.i.A` extension).

The reference listing must be one that is produced by a current release of the TCAT instrumentor. Reference listings produced by earlier versions may not necessarily work correctly with this switch.

If a module is tested but the name is not found in the supplied reference listing, then that coverage is not reported. Similarly, if a name appears in the reference listing and is not one that exists in the archive file, no coverage will be reported.

In case there is an error, `cover` gives a response line (usage line) indicating the set of switches and options. You can also look at the available options by entering `cover -help`. The following will appear on your display:

```
TCAT: Coverage Analyzer. [Release 8.2 for SUN/UNIX 12/17/92]
(c) Copyright 1990 by Software Research, Inc.
```

```
Syntax:  cover [tracefile[s] [options]* = default
-a file  Old archive(* Archive) -n|-N Not Hit Report
-b file  Print title on report  -NH   Newly Hit Report
-c       Cumulative Report     -NM   Newly Missed Report
-d [name[s]] Delete modules named -nl  fileReported module list
-DI      Deinstrumented file    -p   Past Report
-DL      List deinst modules    -q   Quiet Output
-f file  New Archive filename  -r  fileReport file(* Coverage)
-help   Print valid syntax     -s   Sort report by module name
-h [name[s]] Linear Histogram  -SU  Suppress update to archive
-H       Hit Report            -T [#] Threshold value to deinst
-l [name[s]] Logarithmic Histogram -w  widthChange report width
-m       Suppress messages     -Z  fileAnnotated reference listing
```

6.4.1 File Summary

This section describes TCAT file naming conventions for cover.

```
cover [optional switches] [tracefile]
```

Inputs:

Trace.trc (or other file named in execution of program)

Old Archive files

Produces:

Coverage Coverage reports

Archive New archive file which merges latest trace information into cumulative data.

6.4.2 Trace File Argument

The cover command can handle many trace files in the same run. For instance, it is possible to issue the command:

```
cover *.trc -c -n -l ..."
```

to report on all the trace files in the directory with the extension .trc. Of course, one could also issue a command to input data from only one trace file:

```
cover Trace.trc -c -n -l ..."
```

Finally, the Trace.trc file is a default, so the above command is equivalent to the following:

```
cover -c -n -l ...
```

6.4.3 Archive Files

At the end of each run, cover also generates a new archive file that can be used in the next run of cover. The default file name is *Archive*. The archive files created by cover are similar to trace files in their format and content. The significant difference is that they do not contain information on the sequence in which segments were hit. They do, however, contain all other data required for coverage analysis. cover allows you to perform a series of incremental tests. By default, it takes the cumulative summary data stored in the default archive file, *Archive*, produced by previous runs of cover, and submits it as input to the current run of cover. This allows you to add new test suites to exercise unhit segments without having to

include previous test suites. Thus, subsequent test suite size will be smaller.

6.5 ‘mkarchive’ Utility

The *TCAT* system also includes a utility program for creating null archive files. This is *mkarchive*. This utility ensures that your coverage reports all modules on your system whether or not they have been executed. Sometimes, when testing a subsystem, the initial tests do not touch every module in the program. When this occurs, the C1 measure will start at an artificially high level and, as the tests touch more modules, the C1 value will decrease.

Although more segments are being hit, more modules are included in the percentage calculation, so the resulting value is lower. If you are not certain that you can detect whether a module has been skipped over in a lengthy program, it is wise to always use this utility to ensure that your testing coverage data is complete and accurate.

The *mkarchive* utility reads the archive input table *.i.L (Segment Count) file produced by the instrumentation process and creates a “null” archive file containing a complete count of all the modules and their segments in the program being tested. This is a normal archive file and can be used with *cover* to ensure accurate results in generating coverage reports.

To include the *mkarchive* data in your coverage reports, run *mkarchive* before beginning the report generation process with *cover*.

The syntax for *mkarchive* if you have a one file program is:

```
mkarchive < x.i.L > null.arc
```

where *x.i.L* is the archive input table created during instrumentation, and *null.arc* is the null archive file. To use *mkarchive* for multiple files program, concatenate all *.i.L files into one file and execute *mkarchive* on that one file. To include the null archive file in the coverage analysis step, run *cover* with the *-a* option, as in the following example:

```
cover Trace.trc -a null.arc
```

where *Trace.trc* is the trace file.

6.6 Command Summary

This section summarizes commands you use with *TCAT*.

6.6.1 Instrumentation, Compilation and Linking

You are required to preprocess the source file through a C preprocessor before putting it to *ic* instrumentor. The instrumented program is then compiled and linked with the appropriate runtime modules. Depending

on the size of your program and the development method that you use, the following subsections describe how it is done.

Stand-Alone Files

The commands used are:

Preprocess <code>cc -P basename .c</code>	To produce <i>basename.i</i>
Instrument <code>ic basename .i</code>	To produce <i>basename.i.c</i>
Compile <code>cc -c basename .i.c</code>	To produce <i>basename.i.o</i>
Link <code>cc basename .i.o crun1.o</code>	To produce the executable <i>a.out</i>
Execute	Run your program as usual. (Press RETURN twice to accept the default values for trace file message and name.)

Systems With make Files

1. If you have make files where *.o files are created with built-in rules, add the following built-in rule before other targets:

```
# Built in rule for TCAT instrumentation...
.c.o:
cc $(CFLAGS) -P *.c
ic *.i
cc $(CFLAGS) -c *.i.c
mv *.i.o *.o
```

```
sample.o: sample.c
```

```
...
```

```
# The above will depend on which one invokes built in rules.
```

2. Add `crun <level> .o` to the list of linked object modules.
3. Then run the make file to produce the instrumented version of the software.

make Files With cc Called In Directives

When `cc` is explicitly called in directives, then add `ic` commands to the `cc` commands within the make file.

1. Replace `cc w -P filename .c`
`ic filename .i`

```
cc $(CFLAGS) -c filename .i.c
mv filename .i.o filename .o
```

2. Add `crun <level> .o` to the list of linked object modules.
3. Finally, run the make file to produce the instrumented version of the software.

A System Which Does Not Use make Files

(Or which will not allow make file changes)

Go to the directories that contain the source code. There, type the following commands:

```
cc -P *.c
ic *.i
cc -c *.i.c
cc *.i.o crun<?>.o
```

to create the instrumented source, objects and executable.

6.6.2 Program Execution

Run your program as usual.

NOTE: With the default runtimes (runtime level 1), the instrumented program will add two prompts when the first instrumented code is executed. You may fill in a value or press return each time. The prompts may be suppressed by changing the provided runtime. Refer to Chapter 7 for a more detailed description of runtimes available.

6.6.3 Coverage Analysis

Use the command:

```
cover [tracefile] -c -n -h -H -l -NH -NM -p -Z filename .i.A
```

to analyze all reports.

Review the reports produced, add new test cases, repeat whole process. Continue adding tests to your test suites until the C1 coverage value you obtain is acceptable.

Runtime Features

This chapter describes the available runtimes.

LEVEL: This chapter is intended for all users.

7.1 Runtime Descriptions

As mentioned, the test engineer using *TCAT* has a choice of many runtime routines to change the behavior and performance of the instrumented system under test. Different runtimes may be selected by linking in the appropriate module.

Finally, you can write your own runtime package if you need to modify *TCAT* to a particular situation, since the program that is needed is small. For an embedded system where the target system has particular characteristics, rewriting the runtime is a practical way to adapt *TCAT*.

There are a variety of runtime modules for each language.

The function of each runtime package is specified by the format of its name as defined below:

```
<language>run<level>.o "
```

For Example:

```
crun0.o - C, level 0, UNIX
```

TCAT is supplied with three standard runtimes:

crun0 - Raw Trace file ("quiet" runtime)

There is no internal processing or buffering. The trace file is the full, unedited trace of program execution. There is no prompting for trace file name, so the user must indicate the trace file identification at the invocation of the program under test.

crun1 - Standard Trace file

This is the same as **crun0**, but with prompts that ask the user for a test descriptor and the name of the trace file. There is no internal processing or buffering. The trace file is the full, unedited trace of program execution. This is the basic version.

cruna - Multi-Tasking (or forking runtime)

cruna provides for successful data collection when instrumented processes run in parallel. **cruna** is designed for analysis of system calls such as the spawn system command of C. A trace file will be produced for parent and child processes.

7.2 Special Runtimes

NOTE: These runtimes are available as a separate purchase.

crun5 - In-Place Reduction

The C1 statistics of the entire program execution are accumulated in memory. The trace file information is written after the program properly exits.

crun5 allocates enough memory with dynamic memory allocation to do full C1 reduction in place.

crunc - Cross Development

This is source code for runtime which you can cross-compile to use in capturing executions of a cross-compiled executable on a target machine.

The tester will need to adapt the source code of runtime for his/her particular situation. For instance, one alternative with an embedded system is to have the runtime write each trace file record to the development system.

Another alternative is to have each record stored in a file on the embedded system, which is then transferred to the development system.

Customizing TCAT

This chapter explains where the setup information is stored and gives instructions on changing it.

You customize *TCAT* by changing the X Window System resources or setup files. This chapter explains where the setup information is stored and gives instructions on changing it.

Resource files are text files. You can edit them with any standard UNIX text editor. Most of the graphical user interface defaults are set in the *SR* file supplied with the product. It needs to be put in the */usr/lib/X11/app-defaults* directory. If you install *TCAT* using the supplied installation script, the contents of the *SR* file are automatically copied or concatenated to the *SR* file in that directory.

In the following figure is a list of the common GUI defaults. You can change the set defaults by manually changing the *SR* file to avoid resetting GUI parameters every time.

```
SR*geometry: +10+10
!
tcatC*instrument*instrumentFile.directory:
tcatC*instrument*instrumentFile.dirMask: *.c
tcatC*instrument*preprocessorCommand.value: cc -P
tcatC*instrument*preprocessorOptions.value:
tcatC*instrument*preprocessorSuffix.value: .i
tcatC*preprocessSwitch: ON
tcatC*instrument*instrumentorCommand.value: ic
tcatC*instrument*_exitAsKeyword.set: False
tcatC*instrument*exitNotAsKeyword.set: False
tcatC*instrument*deinstrument.set: False
tcatC*instrument*deinstrumentFilename.value: DEINSTRU.fns
tcatC*execute*runtimeObj.directory:
tcatC*execute*runtimeObj.dirMask: crun?.o
tcatC*execute*compileCommand.value: cc -c
tcatC*execute*compileOptions.value: *.i.c
tcatC*execute*linkCommand.value: cc -o
tcatC*execute*linkOptions.value: *.i.o
tcatC*execute*makeCommand.value: make
tcatC*execute*makeOptions.value:
```

```
tcatC*execute*applicationName.value: a.out
tcatC*execute*applicationArguments.value:
tcatC*analyze*traceFilename.dirMask: *.trc
tcatC*analyze*viewSource.dirMask: *.dig
tcatC*analyze*referenceFilename.dirMask: *.i.A
tcatC*analyze*pastTests.set: False
tcatC*analyze*cumulativeTests.set: True
tcatC*analyze*hit.set: False
tcatC*analyze*notHit.set: True
tcatC*analyze*newlyHit.set: False
tcatC*analyze*newlyMissed.set: False
tcatC*analyze*logHistogram.set: False
tcatC*analyze*linearHistogram.set: False
tcatC*analyze*referenceListing.set: False
tcatC*analyze*nonReportModule.set: False
tcatC*analyze*nonReportModuleFilename.value:
tcatC*analyze*thresholdReportModule.set: False
tcatC*analyze*thresholdReportModuleLevel.value: 85
tcatC*analyze*generateFunctionListNotInReport.set: False
tcatC*analyze*noUpdateArchive.set: True
tcatC*analyze*oldArchive.set: False
tcatC*analyze*oldArchiveName.value:
tcatC*analyze*newArchive.set: False
tcatC*analyze*newArchiveName.value:
tcatC*analyze*renameReport.set: False
tcatC*analyze*renameReportName.value:
tcatC*analyze*reportWidth.set: False
tcatC*analyze*reportWidthValue.value:
tcatC*analyze*sortReport.set: False
```

FIGURE 83 TCAT resource file

USER'S GUIDE

S-TCAT

System Test Coverage Analyzer

Ver 8.1



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

ALL RIGHTS RESERVED. No part of this document may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise without prior written consent of Software Research, Inc. While every precaution has been taken in the preparation of this document, Software Research, Inc. assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

TOOL TRADEMARKS: STW, CAPBAK, SMARTS, EXDIFF, TCAT, S-TCAT, TCAT-PATH, T-SCOPE, and TDGEN are trademarks of Software Research, Inc. METRIC is a trademark of SET Laboratories, Inc. and Software Research, Inc. and STATIC is a trademark of Software Research, Inc. and Gimpel Software.

Copyright © 1995 by Software Research, Inc
(Last Update: June 21, 1995)

Introduction

This User Manual is intended for both training and reference. It provides substantial information on *S-TCAT/C*, the System Test Coverage Analysis Tool for the "C" language. This first chapter describes how the User Manual is organized. *S-TCAT/C*, Release 8.2 or later, is supplied with an OSF/Motifstyle graphical user interface.

9.1 Audience

The primary audience for this manual is the software quality assurance tester or development staff who will use *S-TCAT/C* to test newly created or modified software programs written in "C". *S-TCAT/C* is intended for any of the following Software Engineering professionals:

1. The Software Quality Analyst who intends to develop a complete set of tests for a system released by Research and Development. This person should also consider *TCAT/C*, a companion SR coverage analyzer used for logical branch level testing. It measures C1 coverage.
2. The R&D Engineer who wants to test subsystems and module interfaces at the unit or branch level for the highest possible code coverage before product release or submission to the SQA department.
3. The Software Metrics or Independent Evaluation Group that will measure and evaluate the testing of either sub or entire systems. *S-TCAT/C* enables this group to "test the testers". The coverage data might be combined with bug reports, complexity metrics or other data to guide software quality management.

9.2 Purpose

S-TCAT/C can be used for either:

1. Unit testing, where the focus of attention is one or more interconnected modules that will later contribute to a larger system.
2. Measurement of the completeness of a test suite for an entire system consisting of a large number of modules. This is informally known as the "big bang" testing approach. If you are already familiar with some of the ideas of *S-TCAT/C*, you may skip to Chapter 11 for operation details.

9.3 Manual Organization

This User Manual is organized to aid the user, during implementation and for usage. It is divided into the following four sections:

1. Chapter 10 gives a brief overview of *S-TCAT/C* principles. It explains the theories behind *S-TCAT/C* and how it can better help in your testing environment.
2. Chapters 11-14 explain how to use *S-TCAT/C*.
3. Chapters 15-16 explain the appropriate commands, depending on your platform.
4. Chapter 17 displays a step-by-step full *S-TCAT/C* example.
5. Chapter 18 displays a step-by-step graphical user interface tutorial.

Overview

This section provides an overview of coverage analysis principles and of *S-TCAT/C*. It describes how *S-TCAT/C* will fit into the testing phase of the software life cycle.

10.1 Why System Test Coverage Analysis?

The primary purpose of system testing is to ensure the reliability of a software system before it is released to the end user. Mostly, this means making sure that the interfaces between system components are well-exercised, so that latent defects can be removed.

Software should be thoroughly tested with a variety of input to provide statistically verifiable means of demonstrating the product's reliability. In other words, the testing process should cover, in some way, all the situations in which the program will be used. Although a worthy goal, imagining every possible use, as well as developing test data and running them, is extremely complicated and time-consuming. A more realistic goal is to test every interface between components within a system.

According to industry studies, achieving this goal yields significant improvement in overall software quality. Hence, *S-TCAT* improves the quality of your software beyond conventional standards.

10.2 QA Problems Addressed

It is a sad fact of the software engineering world that, on average, without coverage analysis tools, only around 40 percent of a system's interfaces are thoroughly tested before release. With less than half the interfaces actually tried many bugs go unnoticed, and are not revealed until after release. Questions such as when to stop testing, or how much more testing is required are not answered 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 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. Cases testing the same interfaces over and over clutter test suites and take the place of other

cases that would test previously unexplored areas. Often testers are unsure of the direction to take and can waste SQA time devising the wrong (i.e. ineffective) tests.

10.3 Cost Benefit Analysis

S-TCAT/C addresses the problems mentioned above, and can save your organization much time and effort. As a matter of fact, the economics of system interface coverage analysis are extremely favorable. Here are some ways that the *S-TCAT* product can save you money.

10.3.1 Improved Error Detection

Primarily, *S-TCAT/C* provides increased error detection. Software Engineering literature indicates that an average function call error rate is ~6 defects per 1,000 lines of code (KLOC). With no coverage analysis, 40 percent of the function calls are exercised leaving the product with 2.4 defects per KLOC. Assuming a uniform distribution of errors throughout the source code, the simple act of raising the interface coverage rate can uncover many errors.

According to SR's experience in advanced industrial projects and reports from customers, comprehensive interface coverage analysis can eliminate another 80 to 90 percent of the latent software errors.

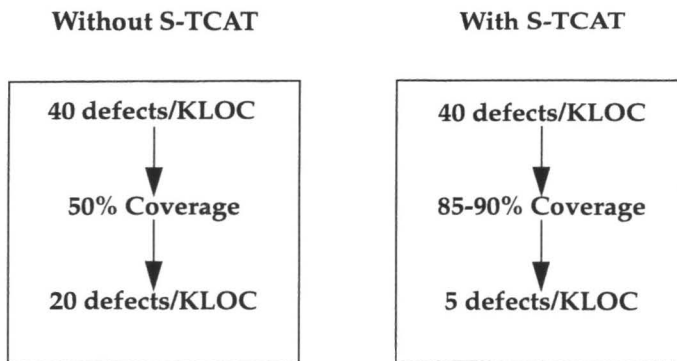


FIGURE 84

Cost Benefit Analysis

The economic value of the increased error detection will, of course, vary considerably from organization to organization. One estimate of the worth of coverage analysis is based on what some 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 around \$4,000 per function call error fixed.

Applying this reasoning to *S-TCAT/C* use, you could save \$9,600 or more per KLOC! In practical terms, this means that a large project with over 20,000 lines of code might save as much as \$192K.

10.3.2 Earlier Error Detection

Not only are more errors detected with *S-TCAT/C*, they are also discovered earlier. It's a well-accepted truth in Software Development that the earlier you catch and fix an error, the cheaper. Over and over, managers, vendors and "software gurus" have shown figures and charts that detail how much less it costs to rectify a defect detected early. A classic example of this is the following, adapted from Barry Boehm's book (see figure below):

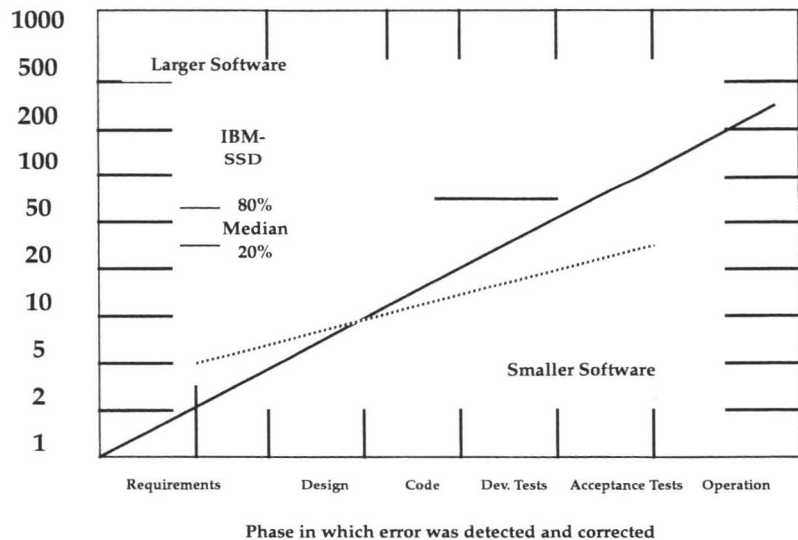


FIGURE 85 Increase in Cost-to-fix Throughout Life-cycle

Your organization can reduce its "cost-to-fix ratio" by a factor of 10 by using *S-TCAT/C* and finding 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 much more inexpensively than the beta site or indepen-

dent testing organization. This is not to say that beta sites or IV&V (Independent Verification and Validation) work are not needed; instead, there is a great cost advantage in letting detailed interface testing find more errors for less cost.

10.3.3 More Efficient Testing

Using *S-TCAT/C*, you can gain in guiding test case development. In general, the product may be used to identify features missed by existing test suites. The missing items then direct the addition of new test cases.

10.3.4 Minimal Test Set

S-TCAT/C 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 will exercise all the function calls of a system, since such test sets typically will require significant time and computing resource to run.

SR recommends use of *SMARTS*, *CAPBAK*, and *EXDIFF* to automate test suite execution, evaluation and analysis steps. These tools can significantly reduce the cost of test suite execution and analysis.

S-TCAT/C can be used to identify and eliminate redundant test cases. With the system interface 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 certain specified minimum coverage threshold, say one percent, for example, it might be reasonable to discard it. Having done so, the tester will achieve a better (i.e. smaller), and thus easier to run, test suite.

10.3.5 Assessment of Progress

Coverage analysis with *S-TCAT/C* 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 $SI > 95\%$ has been set the recommended threshold for proper system interface 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 may wish to take into account are the added test cost and the risk of defects.

Coverage analysis data is 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 product latent defect rates. We encourage SQA managers to keep careful records of errors found and corresponding coverage values.

10.4 Software Test Methods

Interface analysis as implemented through *S-TCAT/C* is a powerful testing technique, which can save you much time and trouble, and can greatly improve software quality. However, it is plainly not the only testing technique in existence. SR strongly recommends that you use *S-TCAT* 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, which are usually performed in their natural sequence, are described below.

10.4.1 Manual Inspection

Programs are manually inspected for conformance to in-house rules (standards) of interface 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".

10.4.2 Dynamic Analysis

This approach tests the dynamic properties of the software under real or simulated operating conditions. The software is executed under controlled circumstances with specific expected results.

It is important in this phase to test as many branches, function calls and paths in the program as possible. Doing so assures that the tests you have run have the greatest diversity-- and hence have the best chance of uncovering defects.

To obtain statistics on the program parts that have been covered by your tests can often be very difficult. Using automated coverage analysis tools such as *TCAT/C*, *S-TCAT/C*, or *TCAT-PATH/C* will produce data on what has been validated and what has been left out of your testing. Dynamic analysis can in aggregate uncover 75 to 90 percent of the latent remaining software defects.

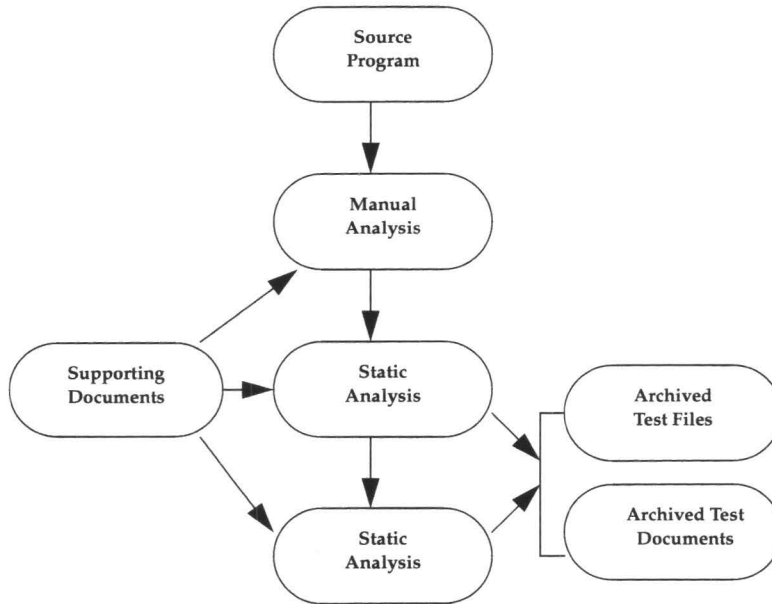


FIGURE 86 Stages in Software Testing

10.5 Multiple-Module Testing

Another consideration in getting the most out of *S-TCAT/C* involves determining the scope of your tests: whether to test a single program module, multiple modules, or even an entire system. You can prepare, or instrument, many modules with function call markers and run tests on them as a group -- *S-TCAT/C* keeps track of each module's level of interface exercise by name.

10.6 Hierarchy of Coverage Metrics

Up to now we have referred briefly to coverage analysis. Let us look more closely at this phrase. What is being covered? What does analysis of results tell you? Interface coverage analysis provides a means of identifying exactly which interfaces in your program systems have been hit, or "exercised", by your tests. The goal is to run your program through a simulation of real operating conditions with many types and combinations of

test data, to explore as many parts of your program as possible. Analysis of the results will lead to more thorough testing and, eventually, to a solid and more reliable software product.

Coverage analysis can be performed at different levels. For example, you can find out which program statements have been hit, or you can analyze the structure of the program by testing which logical branches or segments have been hit. Statement coverage values can vary significantly from logical branch coverage values, depending on properties of the programming language and the programmers' style. *TCAT/C* measures logical branch coverage. An even more rigorous metric involves noting which logical paths have been exercised. *TCAT-PATH* measures path coverage.

When you complete unit level testing, it is appropriate to test system interfaces. In particular, how thoroughly the function calls have been exercised. *S-TCAT/C* provides a functional call completeness measure.

10.7 S1 Measure

The value that measures the level of call-pair coverage is *S1*. This analyzes program testing in terms of the number of function calls -- interfaces from a calling module to a called module -- that are exercised by a test. The *S1* value can be the result of a single test or the accumulated result of a series of combined tests on one or many modules. A definition of *S1* coverage percentage follows:

The percentage of a program's function calls that have been exercised by one or more tests. An *S1* value of 95 percent is a practical minimum coverage level, detecting approximately 75 percent of the then discoverable errors. This high value is usually the accumulated result of a series of tests, since coverage from a single test is only typically 30 percent to 40 percent.

A program is considered 100 percent interface-tested only when every function call has been correctly exercised by at least one test. That is, when *S1* equals 100 percent.

10.8 How Does S1 Relate to C1?

C1 means coverage means the percentage of logical branches exercised during test. Logical segment coverage is an excellent way to measure the completeness of individual module or small groups of module testing. Function call or *S1* coverage describes completeness of the testing of all the interfaces of a complex system. It is important to understand how *S1* measures test completeness. Suppose, for example, that the subroutine calling structure is like that given in the following picture:

```
Sub-A:  
    Sub-B  
    Sub-B  
    Sub-C  
    Sub-D  
    Sub-B
```

```
Sub-B:  
    Sub-C  
    Sub-C  
    Sub-C  
    Sub-D
```

We'll focus on the two topmost modules, Sub-A and Sub-B. Sub-A has three different calls to Sub-B, plus calls to Sub-C and Sub-D. Sub-B has three calls to Sub-C and one call to Sub-D. We will assume that Sub-C and Sub-D do not have any function calls. Our complete system structure contains a total of nine function calls. One test of this system might call Sub-A, which might call Sub-B only and then return. *S-TCAT/C* reports on precisely which function calls are exercised by a test.

S1 coverage analysis is particularly useful when a finished product has been modified. In this case, the logical flow is usually well-tested, although *C1* testing of the modified modules is recommended. However, it is difficult to take into account all the inter-related functions that a modification to the source code may incur.

By using the function tree graph capability of *S-TCAT/C*, (cg, *Xcalltree* utility) one can quickly find which function calls need to be tested. By using *S-TCAT/C* to monitor the actual testing, you can make sure the proper modules are actually tested, thus eliminating errors and guesswork as to whether the modification introduced new errors.

10.9 Advanced Coverage Metrics

There are several other coverage metrics under investigation in industry and research. These metrics incorporate logical segment level coverage and include other logical divisions of the program under test. One metric is "all segments and all boundary conditions for loops", another is "all data paths", that is, all paths between the setting and using of data elements.

One metric is "all segments and all boundary conditions for loops", another is "all data paths"; that is, all paths between the setting and using of data elements.

One metric that includes *C1*, boundary conditions, and all data paths is called *Ct*. *Ct* measures the percentage of all logical paths that are exercised. *Ct* is implemented by SR's *TCAT-PATH*, and according to customer feedback, is ten times more rigorous than *TCAT/C*. In simple terms, programs that have 90 to 100% *C1* coverage typically have 10-15% *Ct* coverage. Please consult the *TCAT-PATH* User's Guide for more information on that utility.



Instrumentation

This and the next three chapters tell how to use *S-TCAT/C* to increase test coverage and detect more software errors.

There are two ways to access *S-TCAT*: from the command line, and with menus. The following command-line invocations are the focus of these chapters.

1. Instrumentation (marking call-pairs)
2. Compiling and Linking with Runtime (recording and counting markers) and Executing
3. Path generation (generating complete path sets)
4. Coverage analysis (reporting call-pairs hit)

A description of how to use the menus appears in Chapter 14.

11.1 Overview

In brief, *S-TCAT/C* instruments the source code of the system to be tested, that is it inserts function calls at each call-pair. The instrumentation will not affect the functionality of the program. When it is compiled, linked and executed, the instrumented program will behave as it normally does, except that it will write coverage data to a trace file. There is some performance overhead due to the data collection process. The trace file is processed by a report generator described later.

Finally, the user looks at the coverage reports to assess testing progress and to plan new test cases. New test cases are added in subsequent passes until a threshold percentage of S1 call-pair coverage has been reached. The coverage reports guide the addition, or possibly the deletion, of tests.

11.2 Instrumentation

As already mentioned, an instrumented program is one that has been specially modified so that, when executed, it transmits information about S1 coverage at every stage of testing while behaving logically equivalent to the original program.

In its operation, *S-TCAT/C*'s instrumentor parses your candidate source code, looking for function call. When one is discovered, the instrumentor inserts a function call in the instrumented version of the source code. It is important to note that the resulting source code file is still a legal program written in "C", as was the original program. The only difference is the added function calls.

When executed, the inserted function calls write to a trace file. Remember, the instrumented version will otherwise function as the uninstrumented version.

11.2.1 The Instrumentor

The complete syntax for command line calls to `s-ic` is listed below.

```
s-ic file.ext [file.ext]  
[-ce]  
[-cw]  
[-DI deinst-file]  
[-fl value]  
[-fn value]  
[-help]  
[-I]  
[-lj]  
[-m]  
[-m6]  
[-n]  
[-t]  
[-u]  
[-w]  
[-x]  
[-z]
```

This command instruments submitted "C" language file(s). It takes `*.i` source file(s) and produces the instrumented file(s): `*.i.c` (for UNIX) or `*.ic` (for MS-DOS or OS/2). `*.c` is the "C" source file, while `*.i` is the preprocessed file.

It is required that the user preprocess the source file through a "C" preprocessor before passing it to `s-ic`. Normally, the preprocessing command is: `cc -P file.c` (for UNIX) or `cl -P file.c` (for DOS running Microsoft C). These commands read `file.c` and produce `file.i`. The options are listed in alphabetical order.

```
file.ext [file.ext]
```

- File(s) to be instrument. **ext** can be "c" or "i". If there are multiple files, then each is processed in the order presented.
- ce** Processes conditional expressions of the form ? a : b .
- cw** Suppresses the "Conditional Expressions Not Processed" warning message.
- DI *deinst-file*** De-instrument Switch. Allows the user to specify a list of modules that are to be excluded from instrumentation. Only the list of module names found in the specified *deinst-file* is to be excluded from instrumentation. The module names can be specified in any format. White space (such as tabs, spaces) is ignored. This switch effects the instrumented (***.i.c**) file and the reference listing (***.i.A**) file.
- fl *value*** Allows the user to specify the maximum length of filename characters that are allowable on the system. If the length of a generated filename exceeds the value, then the instrumentor output will be redirected to files named *Temp.i.?*. These files can be used in subsequent processing.
- fn *value*** The flexname switch. Allows the user to specify the maximum characters of function names the instrumentor recognizes. If the function name exceeds the value, then the instrumentor will recognize as distinct only the first value characters of the function name. For instance, a **-fn 5** will recognize the first five characters as distinct. Characters beyond that point, however, will not be recognized for function name purposes.
- help** Help Switch. Forces output to show a summary of available switches. **NOTE:** This is also the output produced by any illegal command to **s-ic**.
- I** Ignore Errors Switch. In certain rare cases, when the underlying "C" compiler supports non-standard options and constructs, it may be desirable to "force" instrumentation to occur regardless of errors found. This is done with the **-I** switch. **CAUTION:** When instrumentation is forced using this switch, there is a chance that the instrumented software will not compile. For example, if you use the **-I** switch to "instrument" a file of text material, you would not expect the

- output to be compilable (and it probably won't be), even though it may have been "instrumented".
- lj Processes `setjmp` and `longjmp`. This option works only for UNIX.
 - m Recognize Microsoft C 5.1 keywords during the instrumentation process. **NOTE:** This switch applies only to MS-DOS and OS/2 versions. This switch may produce unusual results if used in UNIX systems.
 - m6 Recognize Microsoft C 6.0 keywords during the instrumentation process. **NOTE:** Applies only to MS-DOS and OS/2 versions. This switch may produce unusual results if used in UNIX systems.
 - n Will not instrument empty edges (for example: `if` without `else` or `switch` without `default`.)
 - t Recognize Turbo C keywords during the instrumentation process. **NOTE:** This switch applies only to MS-DOS and OS/2 versions.
 - u Forces the instrumentor to recognize `_exit` as `exit`. **NOTE:** This switch applies only to MS-DOS and OS/2 versions.
 - w Recognize Whitesmith C keywords during the instrumentation process. **NOTE:** This switch applies only to MS-DOS and OS/2 versions.
 - x Will not recognize `exit` as keyword. **NOTE:** This switch applies only to MS-DOS and OS/2 versions.
 - z Recognize MANX/AZTEC "C" keywords during the instrumentation process. **NOTE:** This applies only to MS-DOS and OS/2 versions. This switch may produce unusual results if used on UNIX systems. If there is an error, `s-ic` gives a response line, or usage line, indicating the set of possible switches and options, which is the same as the `-h` output.

11.2.2 Excluding Function Calls from Instrumentation

The *S-TCAT.fns* file contains the list of function calls that are to be excluded from instrumentation. If the user wants to exclude particular functions from instrumentation, he should put those functions in this file.

The *S-TCAT.fns* file can be of any format, as long as the function names are separated by white space. An example of the *S-TCAT.fns* file is supplied with the product and is shown next:

```

assert
atof  atoi  atol
toupper tolower _toupper      _tolower      toascii
ctime  localtime      gmtime  asctime
isalpha isupper islower isdigit isxdigit      isalnum
isspace
        ispunct isprint isgraph iscntrl isascii
cuserid
ecvt fcvt gcv
exit
exp    log    powe   sqrt
fclose fflush
feof  ferror clearerr      fileno
floor ceil   fmod   fabs
fopen freopen fdopen
fread fwrite
frexp ldexp  modf
fseek ftell  rewind
getc  getchar fgetc  getw
getenv
getgrent      getgrnam      setgrent      endgrent
getlogin
getopt
getpwent      getpwuid      getpwnam      setpwent
endpwent
gets  fgets
l3tol ltol3
logname
malloc realloc calloc
mktemp
monitor
nlist
perror
printf fprintf sprintf
putc  fputc  putw

```

For example, `printf` is in the file, so every time `printf` is called in the instrumented module, the instrumentor will not instrument that particular function call.

`my_function` is not in the file, so the instrumentor will instrument every `my_function` function call encountered.

NOTE: In order to use this exclusion feature, *S-TCAT.fns* should reside in your working directory.

11.3 DOS Instrumentation

In DOS you must preprocess before instrumentation. Microsoft C uses the */P* option, Lattice, *-P*. Check your compiler manual for the particulars of the command. Preprocessing may also be accomplished by the 'make' file. An important point about the DOS version of *TCAT* is that some compilers will not accept files that end with *.ic*. It is therefore necessary to rename the program prior to final compilation of the instrumented code.

```
s-ic [optional switches] <filename>.i
```

The above command always produces an instrumented version of the code in a file called *<filename>.ic*. Check for the optional switches available for processing various dialects of "C" such as Turbo C and Microsoft C.

11.4 UNIX Instrumentation

As with DOS, in UNIX you must preprocess your code prior to instrumentation. This task can be accomplished with the following command:

```
cc -P filename.c
```

The preprocessed code can then be instrumented with the following command:

```
s-ic [optional switches] filename.i
```

Instrumentation will create a number of files, one of them being *<filename>.i.c*. It is this file which should be compiled and linked with the appropriate runtime package.

11.4.1 Instrumenting With 'make' Files

Most often, *S-TCAT/C* will be used to develop test suites for systems that are created with 'make' files. Make files cut the time of constructing systems, by automating the various steps necessary to build the system, including compilation and linking.

Fortunately, it is possible to add a few statements to most 'make' files to enable them to make an instrumented version of the system. The modifications fall into two general categories, based on whether or not the make file explicitly names the compiler: *cl* for Microsoft C and *cc* for most UNIX compilers.

If the 'make' file explicitly mentions the "C" compiler with a *cc* command (for example), it is possible to add the *s-ic* command and an

extra `cc` command for preprocessing, instrumenting and compiling causing the make script to instrument and compile the "C" files in question.

Make file lines such as:

```
UNIX:
    sample.o:sample.c
        cc-c sample.c

MS-DOS and OS/2:
    sample.obj:sample.c
        cl /c sample.c
```

would be changed to:

```
UNIX:
    sample.o: sample.c
        cc -P $(CFLAGS) sample.c
        s-ic sample.i
        cc -c $(CFLAGS) sample.i.c
        mv sample.i.o sample.o

MS-DOS and OS/2:
    sample.obj:sample.c
        cl /P $(CFLAGS) sample.c
        s-ic -m6 sample.i
        rename sample.ic temp.c
        cl /c $(CFLAGS) temp.c
        rename temp.obj sample.obj
```

The other situation is where the compiler is not explicitly mentioned, but given as a "built-in" rule. The user can add the following "built-in" rule:

```
UNIX:
    .c.o:
        cc -P $(CFLAGS) $*.c
        s-ic $*.i
        cc -c $(CFLAGS) $*.i.c
        mv $*.i.o $*.o

MS-DOS and OS/2:
    .c.obj:
        cl /P $(CFLAGS) $*.c
        s-ic -m6 $*.i
```



```
rename $*.ic temp.c
cl /c $(CFLAGS) temp.c
rename temp.obj $*.obj
```

The other change necessary is to add SR runtime modules to the link statement. Please refer to Section 12.1-12.2 for more information on the runtime modules.

11.4.2 Example 'make' Files

This section gives several examples of how to create 'make' files that work under MS-DOS and UNIX environments. The first example 'make' file is an illustrative MS-DOS type 'make' file that is unmodified.

```
#####
##
##  S A M P L E    M A K E    F I L E
##  ---W I T H O U T I N S T R U M E N T A T I O N---
##
##
##  DOS version make script for SAMPLE
##
#####
#
OBJS = sample.obj sampley.obj samplel.obj tree.obj ini-
t.obj \
error.obj dotest.obj help.obj log.obj ui.obj premain.obj
license.obj \
pretree.obj preprocl.obj preprocy.obj

CFLAGS = /c /Fpi /AL /DMSDOS /DLIMITED
LFLAGS = /STACK:20000

sample.obj: sample.c

sampley.obj: sampley.c

samplel.obj: samplel.c

tree.obj: tree.c

license.obj: license.c

init.obj: init.c

error.obj: error.c
```

```

dotest.obj: dotest.c

help.obj: help.c

log.obj: log.c

ui.obj: ui.c

premain.obj: premain.c

pretree.obj: pretree.c

preprocl.obj: preprocl.c

preprocy.obj: preprocy.c

sample.exe: $(OBJS)
    sample.obj license.obj help.obj \
        sampley.obj samplel.obj tree.obj init.obj \
        error.obj dotest.obj log.obj ui.obj premain.obj \
        pretree.obj preprocy.obj preprocl.obj \
        link @sample.lnk;

```

FIGURE 87 Uninstrumented DOS Make File

The file below shows the modifications to the 'make' file needed to provide for automatic instrumentation. The modifications are shown in bold face.

```

#####
##
## S A M P L E     M A K E     F I L E
##
## -----W I T H I N S T R U M E N T A T I O N-----
##
##
## DOS version make script for SAMPLE file
##
#####
OBJS = sample.obj sampley.obj samplel.obj tree.obj ini-
t.obj \
error.obj dotest.obj help.obj log.obj ui.obj premain.obj
license.obj \
pretree.obj preprocl.obj preprocy.obj

```

```
CFLAGS = /c /Fp /AL /DMSDOS /DLIMITED
LFLAGS = /STACK:20000
```

```
.c.obj:
    cl $(CFLAGS) /P $*.c
    ic -m6 $*.i
    rename $*.ic temp.c
    cl $(CFLAGS) /c temp.c
    rename temp.obj $*.obj

sample.obj: sample.c

sampley.obj: sampley.c

samplel.obj: samplel.c

tree.obj: tree.c

license.obj: license.c

init.obj: init.c

error.obj: error.c

dotest.obj: dotest.c

help.obj: help.c

log.obj: log.c

ui.obj: ui.c

premain.obj: premain.c

pretree.obj: pretree.c

preprocl.obj: preprocl.c

preprocy.obj: preprocy.c

sample.exe: $(OBJS)
    sample.obj license.obj help.obj \
    sampley.obj samplel.obj tree.obj init.obj \
    error.obj dotest.obj log.obj ui.obj premain.obj
\
```

```
        pretree.obj preprocy.obj preprocl.obj  
crun11.obj \  
        link @sample.lnk;
```

FIGURE 88 Instrumented DOS Make File

The 'make' file below shows a typical UNIX/XENIX 'make' file before modification.

```
#####
##
##  S A M P L E      M A K E      F I L E
##
##  Make file example, no instrumentation.
##
##  UNIX, XENIX
##
#####
# Uses make's knowledge of lex, yacc, cc.
#####
#####

CCextras =
CFLAGS = -s ${CCextras} -DXENIX
YFLAGS = -d
LDFLAGS = -i -ly -ll
LFLAGS = -v
Lextras =
Objects = sample.o sample.y.o sample1.o tree.o init.o
error.o dotest.o log.o \
        ui.o premain.o preprocy.o preprocl.o pretree.o
help.o license.o
Sources = sample.c sample.y.c sample1.c tree.c init.c
error.c dotest.c log.c \
        ui.c premain.c preprocy.c preprocl.c pretree.c
sample.h \
        typedef.h error.h y.tab.h preproc.h help.c
license.c license.h
# UNIX version.  Compiles and links.
sample: $(Objects)
        rm -f sample
        cc $(Objects) $(LDFLAGS) $(Lextras) -o sample
#
sample.y.c: sample.y
        yacc $(YFLAGS) sample.y
        mv y.tab.c sample.y.c
        cp y.tab.h ytab.h
#
sample1.c: sample1.l
        lex $(LFLAGS) sample1.l
        mv lex.yy.c sample1.c
#
preprocy.c: preprocy.y
```

```

yacc $(YFLAGS) preproc.y
cat y.tab.c | sed -e 's/yy/xx/g' > preproc.c
cat y.tab.h | sed -e 's/yy/xx/g' > pretab.h
rm y.tab.c
#
preprocl.c: preprocl.l
lex $(LFLAGS) preprocl.l
cat lex.yy.c | sed -e 's/yy/xx/g' > preprocl.c
rm lex.yy.c
lpr:
pr $(Sources) | lpr

license.o: license.c license.h

```

FIGURE 89 Uninstrumented UNIX Make File

The changes needed have been made in the modified 'make' file shown below. The modifications are shown in bold face.

```

#####
##
##  S A M P L E    M A K E    F I L E
##
##  Make file sample, with S-TCAT/C instrumentation
##
##  UNIX, XENIX
##
#####
# Uses make's knowledge of lex, yacc, cc.
#####

CCextras =
CFLAGS = -s ${CCextras} -DXENIX
YFLAGS = -d
LDFLAGS = -i -ly -ll
LFLAGS = -v
Lextras =
Objects = sample.o sample.y.o sample.l.o tree.o init.o
error.o dotest.o log.o \
          ui.o premain.o preproc.y.o preprocl.o pretree.o
help.o license.o
Sources = sample.c sample.y.c sample.l.c tree.c init.c
error.c dotest.c log.c \
          ui.c premain.c preproc.y.c preprocl.c pretree.c
sample.h typedef.h error.h \
          y.tab.h preproc.h help.c license.c license.h

```

```
# UNIX version.  Compiles and links.
.c.o:
    cc -P $(CFLAGS) *.c
    s-ic *.i
    cc -c $(CFLAGS) *.i.c
    mv *.i.o *.o

#
sample: $(Objects) crun1.o
    rm -f sample
    cc $(Objects) crun1.o $(LDFLAGS) $(Lextras) -o
sample
#
sampley.c: sampley.y
    yacc $(YFLAGS) sampley.y
    mv y.tab.c sampley.c
    cp y.tab.h ytab.h

#
samplel.c: samplel.l
    lex $(LFLAGS) samplel.l
    mv lex.yy.c samplel.c
#
preprocy.c: preprocy.y
    yacc $(YFLAGS) preprocy.y
    cat y.tab.c | sed -e 's/yy/xx/g' > preprocy.c
    cat y.tab.h | sed -e 's/yy/xx/g' > pretab.h
    rm y.tab.c
#
preprocl.c: preprocl.l
    lex $(LFLAGS) preprocl.l
    cat lex.yy.c | sed -e 's/yy/xx/g' > preprocl.c
    rm lex.yy.c
lpr:
    pr $(Sources) | lpr

license.o: license.c license.h
```

FIGURE 90 Instrumented UNIX Make File

11.5 File Summary

This section describes S-TCAT/C file naming conventions for the instrumentor (s-ic).

MS-DOS or OS/2:

s-ic [*optional switches*] *filename.i*

Input:

<filename>.i Preprocessed Source File

Produces:

<filename>.ic Instrumented source
 <filename>.iA Segment Reference Listing
 <filename>.iE Error listing
 <filename>.iL Call-pair count/module
 (Used by **mksarchive**)
 <filename>.iP Call-pair Listing
 (Used by **cg/Xcalltree**)
 <filename>.iS Instrumentation Statistics

UNIX:

s-ic [*optional switches*] *filename.i*

Input:

<filename>.i Preprocessed Source File

Produces:

<filename>.i.c Instrumented source
 <filename>.i.A Call-pair Ref.Listing
 <filename>.i.E Error listing
 <filename>.i.L Call-paircountforeachmodule
 (Used by **mksarchive**)
 <filename>.i.P Call-pair Listing
 (Used by **cg/Xcalltree**)
 <filename>.i.S Instrumentation Statistics

11.6 Embedded Systems

An added benefit resulting from *S-TCAT/C*'s software instrumentation strategy is that the tool may be used with embedded systems. Because *S-TCAT/C*'s output is a syntactically-correct program, the tool can be used on programs that are cross-compiled for target systems. The sequence of steps are: the instrumented code is cross-compiled, linked, then moved to the embedded system.

After execution, coverage data collection occurs on the embedded system, and the trace files are uploaded to the host. The specifics of transferring trace files from the embedded system to the host is dependent on the system in question.

Compiling, Linking and Executing

This chapter explains how to compile, link and execute the instrumented program.

Once instrumentation has been completed, the instrumented version of your "C" program must be compiled and linked with the runtime object modules, sometimes called runtime routines.

The runtime routines are supplied by SR and will write to the trace file. These modules are called from the instrumented code; the added function calls, or "probes", call sub-functions inside the runtime modules.

There are several runtime objects for each computer as described in the next section.

NOTE: Some unreachable code may occasionally be inserted by the instrumentor. This may cause warning messages when compiling, but they are not fatal and the compiler should proceed in spite of them.

12.1 Runtime Descriptions

As mentioned above, the test engineer using *S-TCAT/C* for other languages has a choice of many runtime routines to change the behavior and performance of the instrumented system under test. Different runtimes may be selected by linking in the appropriate module. Some optimize execution speed of the instrumented program, while others decrease the size of the trace file, and still another starts and stops the trace data sampling during execution of the program under test, depending on certain rules that are written in a control file. This is further discussed in the next section. Finally, the user can write his own runtime package if he needs to modify *S-TCAT/C* to a particular situation, since the program that is needed is small.

For an embedded system where the target system has particular characteristics, rewriting the runtime is a practical way to adapt *S-TCAT/C*.

There are a variety of runtime modules for each language. The function of each runtime package is specified by the format of its name as defined below:

```
<language>run<level>.o    (for UNIX)
```

or

```
<language>run<level><model>.obj  (for DOS)
```

Examples:

```
crun0.o      C, level 0, UNIX
frun3.o      Fortran 77, level 3, UNIX
prun2.o      Pascal, level 2, UNIX
crun0m.o     C, level 0, DOS, medium memory model
```

Several versions of runtime are available depending on your needs. This section describes runtimes common to both UNIX and MS-DOS or OS/2 systems. Special runtimes which apply only to UNIX are described in the Section 12.2.

crun0 - Raw Tracefile ("quiet" runtime)

There is no internal processing or buffering. The trace file is the full, unedited trace of program execution. There is no prompting for trace file name, so the user must indicate the trace file identification at the invocation of the program under test.

crun1 - Standard Tracefile

This is the same as **crun0**, but with prompts that ask the user for Test Descriptor and the name of trace file. There is no internal processing or buffering. The trace file is the full, unedited trace of program execution. This is the basic version.

MS-DOS Runtimes

MS-DOS has several runtimes available. You must first determine the memory model you are using for memory management on your system. You will then be able to easily choose from the following list of runtimes. The standard runtimes are **crun1**, while the "quiet" runtimes are **crun0**. Microsoft C has five memory models: S for small; M for medium; C for compact; L for large; and H for huge. Turbo has six memory models: T for tiny; S for small; M for medium; C for compact; L for large; and H for huge.

The following is a partial list of runtimes for MS-DOS, as they appear on the distribution diskette:

```
\RUNTIME\TURBO\STD\CRUN1C.OBJ
\runtime\turbo\std\crun1h.obj
\runtime\turbo\std\crun1l.obj
\runtime\turbo\std\crun1m.obj
\runtime\turbo\std\crun1s.obj
\runtime\turbo\std\crun1t.obj
\runtime\turbo\quiet\crun0c.obj
\runtime\turbo\quiet\crun0h.obj
\runtime\turbo\quiet\crun0l.obj
\runtime\turbo\quiet\crun0m.obj
\runtime\turbo\quiet\crun0s.obj
\runtime\turbo\quiet\crun0t.obj
\runtime\msc51\std\crun1c.obj
\runtime\msc51\std\crun1h.obj
\runtime\msc51\std\crun1l.obj
\runtime\msc51\std\crun1m.obj
\runtime\msc51\std\crun1s.obj
\runtime\msc51\quiet\crun0c.obj
\runtime\msc51\quiet\crun0h.obj
\runtime\msc51\quiet\crun0l.obj
\runtime\msc51\quiet\crun0m.obj
\runtime\msc51\quiet\crun0s.obj
```

NOTE: Microsoft C 5.1 runtimes should be compatible with 6.0 updates.

12.2 Special Runtimes (for UNIX only)

crun2 - In-Place Reduction

The *S1* statistics of the entire program execution are accumulated in memory. The trace file information is written after the program properly exits. **crun2** allocates enough memory with dynamic memory allocation to do full *S1* reduction in place.

crun3 - Multiple Processes

crun3 allows the user to turn on and off trace sampling by changing a control file, */usr/lib/stcat.cntl*. The **crun3** runtime checks the control file after a specified number of trace records have been registered in memory, and writes an archive file if the control file indicates that sampling is to stop and data is to be collected.

The next file contains instructions to control trace sampling. For instance, the first control file statement will cause the instrumented program init to register 1,000 "hits", check the control file and then write the trace file data into an archive file and then stop sampling.

Here is an example of the syntax of the control file: # is a comment.

```
# Here the process named "init" is turned off, but will
# requery # the 'stcat.control' file after 1000 segment
# hits: init -1000
# Here the process named "my.oracle" is turned on, and
# will # will requery the 'stcat.control' file after 25000
# segment hits: my.oracle +25000
# Here, the process "trick" has been told to record essen
# tially forever, and "bad" has been told to not record
# test data # essentially forever:
trick +50000000000
bad -50000000000

# Caution: multiple continuously executing instrumented
# programs will always check the 'stcat.control' file on
# startup. If their # name is NOT found anywhere, then
# they will NEVER requery the # 'stcat.control' file
# again.
```

cruna - Multi-Tasking (or forking runtimes)

cruna provides for successful data collection when instrumented processes run in parallel.

cruna is designed for analysis of system calls such as the "spawn" system command of "C". A trace file will be produced for parent and child processes.

crunc - Cross Development

Available as a separate purchase. This is source code for **crun0-3**, which you can cross-compile to use in capturing executions of a cross-compiled executable on a target machine. The tester will need to adapt the source code of runtime for his/her particular situation. For instance, one alternative with an embedded system is to have the runtime write each trace file record to the development system.

Another alternative is to have each record stored in a file on the embedded system, which is then transferred to the development system.

12.3 Executing the Instrumented Program

The next step is to run your instrumented program and track which function calls have been exercised by the test data you supply. In essence, this is a matter of noticing the not-hit call-pairs mentioned in the **Not Hit** report, and looking up the corresponding code in the **Reference Listing**.

S-TCAT/C senses when call-pairs are hit by monitoring the markers inserted during instrumentation and by accumulating the results in a trace file and an archive file, which then becomes the basis for all subsequent *S-TCAT/C* coverage reports.

To produce the trace file, first run your instrumented and compiled "C" program and follow the *S-TCAT/C* prompts. If you use the standard runtime routines, the system will respond with:

""Type in a description of the test run. Be as descriptive as needed for your own information in referring to this test run. You can enter up to 80 characters of text in your message. This message will be recorded in the trace file and used in scover reports. If you choose to enter no descriptive text, just press the return key.

The system next will prompt you for an output filename:

```
Name of tracefile [default is Trace.trc]:
```

Type in any name. The system will create a trace file with the name you enter. To use the default name *Trace.trc*, just press the return key. The trace file description and name are useful in keeping track of different test runs. Consistent, clear naming conventions are useful in organizing different groups of results. A common practice is to identify trace files with the filename extension *.trc*.

Performance Considerations

Sometimes, an instrumented program will produce very large trace files. One solution to this is to compile a mixture of instrumented and uninstrumented files so that the program is tested in pieces.

Coverage Reporting and Analysis

To get useful results from *S-TCAT/C*, you must analyze coverage reports. To do this, the program **scover** is run to process the trace file and produce several output reports. In general, the reports give the following information:

1. Reports included in the current report.
2. A summary of past coverage runs.
3. Current and cumulative coverage statistics.
4. A list of call-pairs that have been hit.
5. A list of call-pairs that have been missed.
6. Bar charts of the frequency of execution of each call-pair.

These reports are useful for performance analysis and also for "hot spot" tuning. The two types of graphs, called histograms, show the frequency distribution of call-pairs hit on either linear or logarithmic scales.

scover also archives the trace file information into an Archive file so that the reports are cumulative. The diagram in Figure 91 shows the components and interfaces of the system.

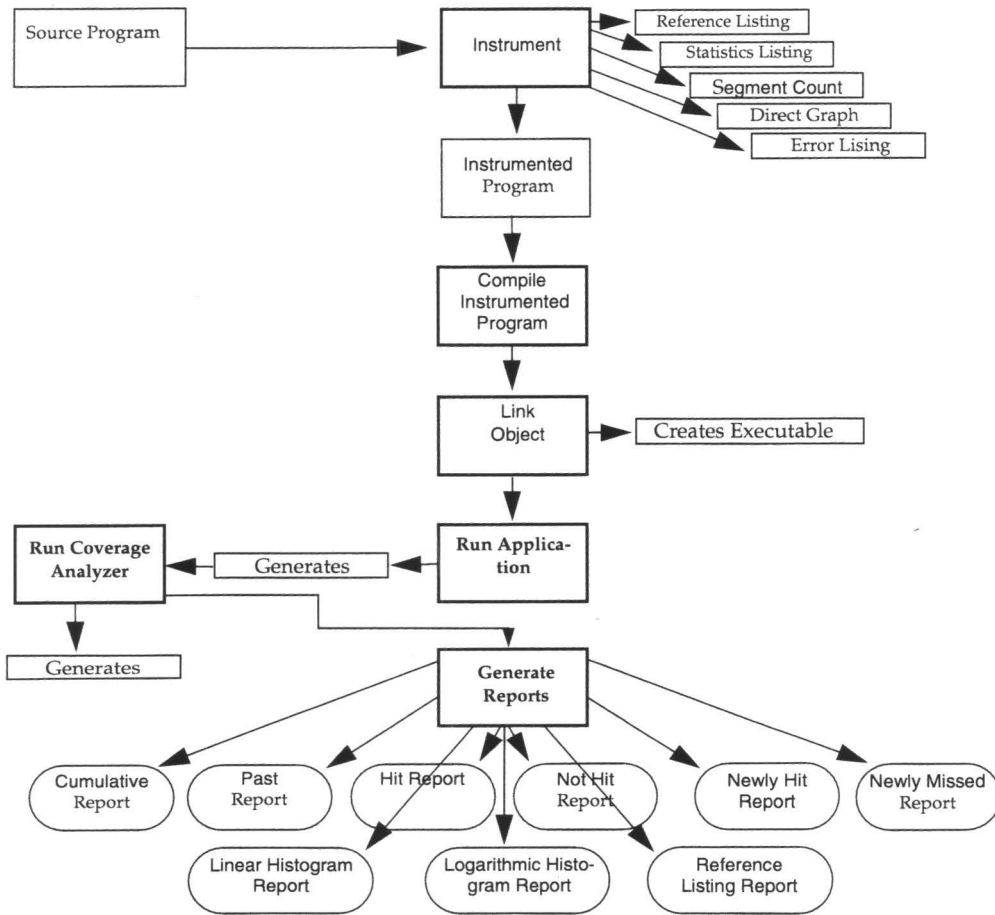


FIGURE 91 System Components

13.1 Producing Reports

This section is an in-depth reference on **scover** and the reports it produces.

The **scover** command analyzes trace files produced by instrumented programs and generates a set of coverage reports.

13.1.1 Report Types

Reports generated by **scover** are stored by default in the file *Coverage*. Depending on the options used, **scover** produces different reports. The reports accomplish one or more of the following:

1. Summarize the S1 coverage achieved by current and cumulative tests on a module by module bases. This is the **Cumulative Report**.
2. Indicate which call-pairs have been hit and which have been ignored by your test cases. These are the **Hit** and **Not Hit** reports.
3. Analyze a current or a past test suite execution. The particular runs to be investigated are selected by choosing the appropriate trace and archive files. The trace file contains information from the most recent run, and archive file contains information from previous runs. This is the **Past Report**.
4. Indicate which call-pairs were hit in the current execution which were not hit previously and vice versa. These are the **Newly Hit** and **Newly Missed** reports.
5. Examine how often call-pairs in a module have been exercised. This is a performance analysis at the function call level. The **Logarithmic Histogram** and **Linear Histogram** reports are the reports of interest here.

13.1.2 Trace File Argument

The **scover** command can handle many trace files in the same run. For instance, in UNIX it is possible to issue the command:

```
scover *.trc -c -n -l ...
```

to report on all the trace files in the directory with the *extension.trc*. Of course, one could also issue a command to input data from only one trace file:

```
scover Trace.trc -c -n -l ...
```

Finally, the *Trace.trc* file is a default, so the above command is equivalent to the following:

```
scover -c -n -l ...
```

13.1.3 Archive Files

At the end of each run, **scover** also generates a new archive file that can be used in the next run of **scover**. The default filename is *Archive*. The archive files created by **scover** are similar to trace files in their format and content. The significant difference is that they do not contain information on the sequence in which call-pairs were hit. They do, however, contain all other data required for coverage analysis.

scover allows the user to perform a series of incremental tests. By default, it takes the cumulative summary data stored in the default archive file, *Archive*, produced by previous runs of **scover**, and submit it as input to the current run of **scover**. This allows the user to add new test suites to exercise unhit call-pairs without having to include previous test suites. Thus, subsequent test suite size will be smaller.

13.1.4 'scover' Syntax

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

```
scover [tracefile [tracefile]]
        [-a old-archive ]
        [-b title ]
        [-c]
        [-d name [name] ]
        [-DI deinst-file ]
        [-DL]
        [-f new-archive ]
        [-help]
        [-h | -h name [name] ]
        [-l | -l name [name] ]
        [-H]
        [-NH]
        [-NM]
        [-m]
        [-N]
        [-n]
        [-nl namefile ]
        [-p]
        [-q]
        [-r report ]
        [-s]
        [-SU]
```

```
[-T [ threshold ] ]
[-w width ]
[-Z reference listing ]
```

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

[tracefile [tracefile]]

These are the names of the tracefiles that you wish to process. If there are no trace files given, then *scover* 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 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 *scover* 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 *title* Banner File Name Switch.

This allows you to include specific text, taken from the first line of the named file, *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.

-d *name [name]* Module Name Delete Switch. If this switch is present then the named modules, if found in the current exe-

cution, are deleted from the generated Archive file. Subsequently, `scover` 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 `-T` switch) will be written to.
- 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 don't include a different name with this switch, the accumulated test data will be placed in the default name `Archive`. **CAUTION:** Each time you run `scover`, 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.
- help** Help Switch. Forces output to show a summary of available switches. Note: This is also the output produced by an illegal command.
- h | -h name [name]**

Linear Histogram Report Switch (`-h`).
- l | -l name [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 data in-

cluded 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, `scover` 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, `scover` 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 effected 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 produced individual histogram reports.

- NM** Newly Missed Report Switch.

Shows which segments, by module, hit in any prior test but 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 scover 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.
- s** Sort Switch. This option produces output reports with module names sorted alphabetically.
- SU** Suppress Update Switch.

During processing, scover will suppress updating of the archive file, either the default Archive or the file named by the -f switch. scover 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 the threshold value. Any module with percentage coverage greater than or equal to this threshold value, will be written

to the "de-instrumented" file. If no threshold is specified, then the default value of 85 percent is assumed.

-w

width Report Width Switch.

Normally the reports generated by `scover` 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 `scover` 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.

-z

reference listing Annotated Reference Listing Switch. `scover` will analyze the specified archive file, any specified trace files, and will produce a report that shows the coverage level achieved for all modules that are named in the specified reference listing (file with `.i.A` or `.iA` extension). The reference listing must be one that is produced by a current release of the `TCAT/C` instrumentor. Reference listings produced by earlier versions may not necessarily work correctly with this switch.

If a module is tested but the name is not found in the supplied reference listing, then that coverage is not reported. Similarly, if a name appears in the reference listing and is not one that exists in the archive file, no coverage will be reported.

In case there is an error, `scover` gives a response line (usage line) indicating the set of switches and options.

13.2 'mksarchive' Utility

The `S-TCAT/C` system also includes a utility program for creating null archive files. This is `mksarchive`. This utility ensures that your coverage reports all modules on your system whether or not they have been executed. Sometimes, when testing a subsystem, the initial tests do not touch every module in the program. When this occurs, the `S1` measure will start at an artificially high level and, as the tests touch more modules, the `S1` value will decrease. Although more call-pair are being hit, more mod-

ules are included in the percentage calculation, so the resulting value is lower.

Most experienced *S-TCAT/C* users are aware of this phenomenon and use the `mksarchive` utility to monitor the total could-have-been-hit count. If you are not certain that you can detect whether a module has been skipped over in a lengthy program, it is wise to always use this utility to ensure that your testing coverage data is complete and accurate.

The `mksarchive` utility reads the archive input table `*.i.L` or `*.iL` (Call-pair Count) file produced by the instrumentation process and creates a "null" archive file containing a complete count of all the modules and their call-pairs in the program being tested. This is a normal archive file and can be used with `scover` to ensure accurate results in generating coverage reports.

To include the `mksarchive` data in your coverage reports, run `mksarchive` before beginning the report generation process with `scover`.

The syntax for `mksarchive` if you have a one file program is:

```
mksarchive < x.i.L > null.arc (for UNIX)
```

or

```
mksarchi < x.iL > null.arc (for DOS)
```

where `x.i.L` is the archive input table created during instrumentation, and `null.arc` is the null archive file. To use `mksarchive` for multiple files program, concatenate all `*.i.L` files into one file and execute `mksarchive` on that one file. To include the null archive file in the coverage analysis step, run `scover` with the `-a` option, as in the following example:

```
scover Trace.trc -a null.arc"
```

where `Trace.trc` is the trace file.

13.3 File Summary

This section describes *S-TCAT/C* file naming conventions for `scover`.

`scover [optional switches] [tracefile]`

Input:

Trace.trc (or other file named in execution of program)

Old Archive files

Produces:

Coverage

Coverage report

Archive

New archive file which merges latest trace information into cumulative data.



Menus

The second way to access *S-TCAT* is with menus, and this chapter will explain how to do so. If you would rather use command-line invocation, you may skip this chapter and go on to Chapters 15,-16, or the full *S-TCAT* example in Chapter 17.

14.1 S-TCAT/C ASCII Menus

The *S-TCAT* ASCII menus and their use are described below. Menu help users in two ways: by providing a fixed structure for collecting test coverage information and by providing a convenient way to customize a sequence of operations.

14.1.1 Invoking S-TCAT

Start up *S-TCAT* in interactive mode with the command:

```
stcat [-r file]
```

where, *file* is the optional configuration file (rc file) name.

The default name for the configuration file is *stcat.rc*. If you don't specify a configuration file, or if *S-TCAT* doesn't find the file *stcat.rc* in the current directory, then *S-TCAT* issues a warning message and continues processing, using default values. Remember that the content of the *S-TCAT* configuration file, *stcat.rc*, always overrides the internally supplied (default) values of all parameters.

14.1.2 S-TCAT Menu Tree

The menu tree is shown in the diagram below.

```
S-TCAT

MAIN:
|
|   Selects ACTIONS or FILES or OPTIONS menus
|   Shows option settings
|   Shows current execution stats
|   Saves option settings
|   Exit from S-TCAT system
|   On-line help frames
|   !<system commands>
|
|___ACTIONS:
|
|   Selects basic S-TCAT operations
|   Shows option settings
|   Return to prior menu.
|   On-line help frames
|   !<system commands>
|
|___OPTIONS:
|
|   Helps select all user-settable options
|   Shows option settings
|   Return to prior menu.
|   On-line help frames
|   !<system commands>
|
|___FILES:
|
|   Shows all current file settings
|   Allows changing file settings
|   Return to prior menu.
|   On-line help frames
|   !<system commands>
```

After *S-TCAT* starts, you will see the title information, version control indication, and the prompt "S-TCAT:MAIN:". To see the available menu options, type from any prompt within *S-TCAT*:

```
? and then [RETURN].
```

S-TCAT then displays the available options for that menu. This feature works for all menus throughout *S-TCAT*. The current menu is redrawn whenever you give an unrecognized command.

Issuing Commands

You can issue commands by typing the first few letters of each command's name. The only requirement is that the letter sequence be unique to that command. *S-TCAT* will inform you when a command you issue matches two or more possible commands. To set variables (see the options menu description, below) you must type the entire variable name. This is done in order to be consistent with configuration file processing.

Displaying Current Parameter Settings

You can display the current settings (options and filenames) known to *S-TCAT* at any time using the settings command, get on-line help with the help command, and exit the current menu using exit. The configuration file reading in the settings is automatically used. However, the settings can be changed if required.

S-TCAT Menu 'Stack'

You can move from the MAIN menu to any other menu at will. *S-TCAT* remembers the sequence of your choice of menus in an internal "stack". This means that when switching from one menu to another, you can return to the immediately prior menu with the exit command. This feature is provided to prevent you from entering conflicting or incorrect data during a run.

If you wish, you can issue a series of exit commands that will eventually return you to the MAIN menu to exit the system. That is, your moves between the three subsidiary menus are "stacked" and must be "unstacked" before returning to the MAIN menu. If you press the DEL key, you return immediately to the MAIN menu.

14.1.3

MAIN Menu

When you invoke *S-TCAT*, the following menu is displayed:

```
S-TCAT:MAIN:
Options:
    actions-- Go to the ACTIONS menu
    files-- Go to the FILES menu
    options-- Go to the OPTIONS menu

    settings-- List current settings for S-TCAT
               options
    help [opt]-- Display HELP text for a command
    release-- Show release and version number
```

```
save-- Save the current settings for S-TCAT
exit-- Exit from S-TCAT to system
```

14.1.4 ACTIONS Menu

The ACTIONS menu is displayed below:

```
S-TCAT:ACTIONS:
Options:
    preprocess-- Run the preprocessor on designated module
    instrument-- Run S-TCAT instrumentor on designated module
    compile-- Execute standard compilation step
    link-- Execute standard linkage step
    make-- Execute specified make command
    go-- Execute instrumented program
    scover-- Execute S-TCAT Coverage Analyzer
    view-- View S-TCAT Coverage Report

    files-- Go to the FILES menu
    options-- Go to the OPTIONS menu

    settings-- Display current runtime settings
    help [opt]-- Display HELP text for command
    release-- Show release and version number
    exit-- Exit current level
```

FILES Menu

The FILES menu is displayed below:

```
S-TCAT:FILES:
Options:
    prefix <name>-- Base name ('prefix') of file processed
    tracefile <name>-- Name of trace file (def.= Trace.trc)
    archive <name>-- Name of trace file (default Archive)
    report <name>-- Name of report file (def. Coverage)

    actions-- Go to the ACTIONS menu
    options-- Go to the OPTIONS menu

    settings-- Display current runtime settings
    help [opt]-- Display HELP text for command
    release-- Show release and version number
    exit-- Exit current level
```

If you change the configuration file from this menu, the *stcat.rc* file (or the file you specified on invoking *S-TCAT*) is not automatically updated. When you exit, *S-TCAT* will prompt you about saving the current settings.

14.1.5 OPTIONS Menu

The OPTIONS menu is displayed below:

```
S-TCAT:OPTIONS:
Options:
    preprocess-- Specify the preprocessor command
    instrument-- Specify the instrument command
    compile-- Specify the compiler command
    link-- Specify the linker command
    make-- Specify the make command
    execute-- Specify the 'go' command
    scover-- Specify coverage analyzer command
    view-- Specify view command for cov. report

    actions-- Go to the ACTIONS menu
    files-- Go to the FILES menu

    settings-- Display current runtime settings
    showmenu-- Toggle showmenu option on and off
    help [opt]-- Display HELP text for a command
    release-- Show release and version number

    exit-- Exit to the system
```

14.1.6 Saving Changed Option Settings

Before leaving *S-TCAT*, the user will be prompted to save the current settings (unless this has already been done in the current execution of *S-TCAT* and the options have not been changed since they were last saved).

Upon exiting *S-TCAT*, you are prompted:

```
Do you want to save current parameter settings (y/n): y
Do you want to use default filename (stcat.rc) (y/n): n
Specify filename: example.rc
Parameter settings saved in example.rc.
```


14.1.7 Running System Command

You may issue a command directly to the operating system by using the **!** symbol, as follows:

```
S-TCAT:!<any system command>
```

S-TCAT regains control after the command is executed. This feature is useful for editing files and for other activity within an *S-TCAT* session.

14.2 S-TCAT Configuration File

This chapter describes the *S-TCAT* configuration file. A sample file is shown at the end of this chapter. The *S-TCAT* menu system reads the configuration file before starting processing. This file can contain modifications to the default settings of a variety of *S-TCAT* parameters. The user can specify which file to use, or *S-TCAT* will automatically use the default name *stcat.rc*. This feature allows the user to set various run-time parameters automatically. Command-line parameters, however, always override the configuration file settings whenever command-line parameters are present.

The *S-TCAT* configuration file is a simple ASCII text file that can be created with an editor. Alternatively, you can create this file, by using the save option from within an interactive invocation of *S-TCAT*.

14.2.1 Configuration File Syntax

The following run-time parameters can be set in the configuration file. Configuration file lines can contain any set of commands in any order. Comment lines must begin with a **#** as the first character. All white space (tabs and blanks) are ignored, except those appearing within quotes.

The latest occurring command prevails in the case of duplicate commands. This feature may be useful when handling several configuration files that differ only slightly.

*<comment>* A line beginning with **#** is treated as a comment.

help=*<filename>* This parameter defines the location of the on-line helpframe information used by *S-TCAT*. Normally it does not have to be re-set if the file of help information is placed at the 'standard' location.

preprocess=*<text>*

This is the text of the command to be used to preprocess the file whose prefix name is given below.

prefix=*<name>* This is the "basename" for the file you are processing. *S-TCAT* automatically adds the appropriate suffix to

indicate the kind of file it is. For example, for a "C" program the suffix is.c.

`tracefile=<filename>`

This filename is the one that is assumed to be used as a trace file.

`report=<filename>`

This filename is the one that is assumed to be used for coverage reports.

`instrument=<text>`

This is the text of the command used to instrument the file whose prefix is given in the system settings.

`compile=<text>` This is the text of the command used to compile the instrumented program.

`link=<text>` This is the text of the command used to link the instrumented program with the runtime package.

`make=<text>` This is the command text to run when the make command is run.

`execute=<text>` This is the text of the command to use to execute the instrumented program.

`scover=<text>` This is the text of the command to use, including any switches that might be needed, to analyze the named trace file.

`view=<text>` This is the command to use to review the Coverage Report.

`archive=<filename>`

This is the filename to use as the Archive File (permanent test record).

`showmenu`

`noshowmenu`

These switches determine whether the entire menu is re-drawn on the screen when a command is issued. You will probably prefer to use noshowmenu after you are familiar with the program.

14.2.2 Sample S-TCAT Configuration File

Below is an example of a typical *S-TCAT* configuration file.

```
#
# Example of S-TCAT Configuration File.
#
noshowmenu
help="/usr/lib/stcat/stcat.hlp"
preprocess="cc -P"
instrument="s-ic"
prefix="example"
report="Coverage"
link="cc *.i.o crun1.o"
compile="cc -c *.i.c"
make="make"
execute="a.out"
scover="scover -n -h"
view="vi"
tracefile="Trace.trc"
archive="Archive"
```

Command Summary: MS-DOS, OS/2

This chapter gives a short command summary for *S-TCAT/C* running under MS-DOS or OS/2.

15.1 Instrumentation, Compilation and Linking

The user is required to preprocess the source file through a "C" preprocessor before putting it to **s-ic** instrumentor. The instrumented program is then compiled and linked with the appropriate runtime module.

Depending on the size of your program and the development method used, the following subsections describe how it is done.

15.1.1 Stand-Alone Files

Here are the commands you would use with the Microsoft C 6.0 compiler on MS-DOS or OS/2:

```
Preprocess: cl /P <filename>.c /* to produce <filename>.i */
Instrument: s-ic -m6 <filename>.i /* to produce <filename>.ic */
Compile: cl /c /Tc <filename>.ic /* to produce <filename>.obj */
Link: cl <filename>.obj crun1s.obj /* to produce <filename>.exe */
```

Execute: (Run your program as usual. Press RETURN twice to accept the default values for trace file message and name.)

Note that **-m6** is the **s-ic** switch for Microsoft C 6.0 compiler. **/Tc** is a Microsoft C 6.0 option that allows for compilation of files with extensions other than **.c**.

Also, note that **crun1s.obj** is the runtime object module that comes with *S-TCAT/C*. There are various runtime object files, depending on compiler, runtime level, and memory model used. For more runtime descriptions on MS-DOS runtimes, turn to Section 12.1.

15.1.2 Systems With 'make' Files

1. In systems that have 'make' files where *.obj* files are explicitly listed as targets, add the following built-in rule before other targets:

```
# Built in rule for S-TCAT instrumentation...
.c.obj:
    cl $(CFLAGS) /P $*.c      cl. $(CFLAGS) $*.c
    s-ic -m6 $*.i           or   s-ic -m6 $*.i
    ren $*.i temp.c         cl $(CFLAGS) /c/Tc $*.ic
    cl $(CFLAGS) /c temp.c
    ren temp.o $*.obj

sample.obj: sample.c
    ...
```

2. Add `crun<level><model>.obj` to the list of linked object modules. You must choose the version of runtime to use, based on the runtime level and the memory model (small, compact, medium, large or huge).
3. Run the 'make' file to produce the instrumented program.

15.1.3 'make' With 'cl', 'msc'

This section deals with situations that involve 'make' files for commonly available PC-based compilers, such as Microsoft C, where compile statements are explicitly mentioned.

1. Replace 'cl' (or 'msc') with the following lines:

```
cl $(CFLAGS) /P <filename>.c
s-ic -m6 <filename>.i
ren <filename>.i temp.c
cl $(CFLAGS) /c temp.c
ren temp.o <filename>.o
```
2. Add `crun<level><model>.obj` to the list of linked object modules.
3. Run the make file to produce the instrumented program.

15.1.4 Systems Without 'make' Files

Go to the directories with the source code and follow the method for stand alone files with each source code file (preprocess, instrument, compile). Finally, link all the object files with the appropriate runtime object file.

15.1.5 Program Execution

Run your program as usual.

NOTE: With the default runtimes (runtime level 1), the instrumented program will add two prompts when the first instrumented code is executed. You may fill in a value or press return each time. The prompts may be suppressed by changing the provided runtime. Refer to Section 12.2 for a more detailed description of runtimes available.

15.2 Coverage Analysis

Use the command:

```
scover [tracefile] -p -c -H -N -h
```

to analyze reports.

Review the reports produced, add new test cases, repeat whole process. Continue adding tests to your test suites until the S1 coverage value you obtain is acceptable. This is a general coverage reporting. For more information, refer to Chapter 13.

Command Summary-UNIX

This chapter summarizes commands you use with *S-TCAT/C* in UNIX and UNIX-like environments.

16.1 Instrumentation, Compilation and Linking

The user is required to preprocess the source file through a "C" preprocessor before putting it to *s-ic* instrumentor. The instrumented program is then compiled and linked with the appropriate runtime modules. Depending on the size of your program and the development method that you use, the following subsections describe how it is done.

16.1.1 Stand-Alone Files

The commands used are:

```
Preprocess:  cc -P <filename>.c /* to produce <filename>.i */
Instrument:  s-ic <filename>.i /* to produce <filename>.i.c */
Compile:    cc -c <filename>.i.c /* to produce <filename>.i.o */
Link:       cc <filename>.i.o crun1.o /* to produce a.out */
```

```
Execute:    (Run your program as usual. Press RETURN
            twice to accept the default values for
            trace file message and name.)
```

16.1.2 Systems With 'make' Files

1. If you have 'make' files where *.o files are created with built-in rules, add the following built-in rule before other targets:

```
# Built in rule for S-TCAT instrumentation...
.c.o:
    cc $(CFLAGS) -P $*.c
    s-ic $*.i
    cc $(CFLAGS) -c $*.i.c
    mv $*.i.o $*.o

sample.o: sample.c
    ...
# The above will depend on which one invokes built in rules.
```


2. Add `crun<level>.o` to the list of linked object modules.
3. Then run the 'make' file to produce the instrumented version of the software.

16.1.3 'make' files with cc called in directives

When `cc` is explicitly called in directives, then add `s-ic` commands to the `cc` commands within the 'make' file.

1. Replace `cc` with the following lines:

```
cc $(CFLAGS) -P <filename>.c
s-ic <filename>.i
cc $(CFLAGS) -c <filename>.i.c
mv <filename>.i.o <filename>.o
```

2. Add `crun<level>.o` to the list of linked object modules.
3. Finally, run the make file to produce the instrumented version of the software.

16.1.4 A system which does not use 'make' files

(Or which will not allow 'make' file changes)

Go to the directories that contain the source code. There, type the following commands:

```
cc -P *.c
s-ic *.i
cc -c *.i.c
cc *.i.o crun<?>.o
```

to create the instrumented source, objects and executable.

16.2 Program Execution

Run your program as usual.

NOTE: With the default runtimes (runtime level 1), the instrumented program will add two prompts when the first instrumented code is executed. You may fill in a value or press return each time. The prompts may be suppressed by changing the provided runtime. Refer to Section 12.2 for a more detailed description of runtimes available.

16.3 Coverage Analysis

Use the command:

```
scover [tracefile] -p -c -H -N -h
```

to analyze reports.

Review the reports produced, add new test cases, repeat whole process. Continue adding tests to your test suites until the *S1* coverage value you obtain is acceptable. This is a general coverage reporting. For more information, refer to Chapter 13.

Full S-TCAT Example

This chapter describes a full *S-TCAT* example, that includes a sample “C” program, instrumented program and referenced listing.

17.1 Introduction

It is assumed that *S-TCAT/C* will be used on syntactically correct programs, that is programs that will compile cleanly before instrumentation. Of course, *S-TCAT/C* will be used to verify that each program segment or function call executes correctly under typical operating conditions. Figure 92 show a sample “C” program with three function modules.

This example program will be used throughout the chapter to describe each component of *S-TCAT/C* to better aid the user.

```

/* EXAMPLE.C --example file for use w. TCAT, STCAT, TCAT-PATH. */
#include "stdio.h"
#include <ctype.h>

#define INPUTERROR      -1
#define INPUTDONE      0
#define MENU_CHOICES   13
#define STD_LEN        79
#define TRUE           1
#define FALSE          0
#define BOOL           int
#define OK              TRUE
#define NOT_OK         FALSE

char menu[MENU_CHOICES][STD_LEN] = {
    "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
    "    What type of food would you like?\n",
    "\n",
    "    1      American 50s   \n",
    "    2      Chinese      - Hunan Style \n",
    "    3      Chinese      - Seafood Oriented \n",
    "    4      Chinese      - Conventional Style \n",
    "    5      Danish        \n",
    "    6      French        \n",

```

```
        "      7      Italian      \n",
        "      8      Japanese     \n",
        "\n\n"
};
int char_index;

main(argc,argv)      /* simple program to pick a restaurant */
int   argc;
char  *argv[];
{
    int i, choice, c,answer;
    char str[STD_LEN];
    BOOL ask, repeat;
    int proc_input();
    c = 3;
    repeat = TRUE;
    while(repeat) {
        printf("\n\n\n");
        for(i = 0; i < MENU_CHOICES; i++)
            printf("%s", menu[i]);
        gets(str);
        printf("\n");
        while(choice = proc_input(str)) {
            switch(choice) {
                case 1:
                    printf("\tFog City Diner  1300 Battery  982-2000 \n");
                    break;
                case 2:
                    printf("\tHunan Village Rest839 Kearney 956-7868 \n");
                    break;
                case 3:
                    printf("\tOcean Restaurant726 Clement 221-3351 \n");
                    break;
                case 4:
                    printf("\tYet Wah 1829 Clement  387-8056 \n");
                    break;
                case 5:
                    printf("\tEiners Danish Res. 1901 Clement386-9860 \n");
                    break;
                case 6:
                    printf("\tChateau Suzanne 1449 Lombard  771-9326 \n");
                    break;
                case 7:
                    printf("\tGrifone Ristorante 1609 Powell397-8458 \n");
                    break;
                case 8:
                    printf("\tFlints Barbecue  4450 Shattuck, Oakland \n");
                    break;
                default:
```

```
        if(choice != INPUTERROR)
            printf("\t>>> %d: not a valid choice.\n", choice);
        break;
    } }
for(ask = TRUE; ask; ) {
    printf("\n\tDo you want to run it again? ");
    while((answer = getchar()) != '\n') {
        switch(answer) {
            case 'Y':
            case 'y':
                ask = FALSE;
                char_index = 0;
                break;
            case 'N':
            case 'n':
                ask = FALSE;
                repeat = FALSE;
                break;
            default:
                break;
        } } } }

int proc_input(in_str)
char *in_str;
{
    int tempresult = 0;
    char bad_str[80], *bad_input;
    BOOL got_first = FALSE;
    bad_input = bad_str;

    while(!isspace(in_str[char_index]))
        char_index++;
    for( ; char_index <= strlen(in_str); char_index++) {
        switch(in_str[char_index]) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                /* process choice */
                tempresult = tempresult * 10 + (in_str[char_index] - '0');
                got_first = TRUE;
                break;
        }
    }
}
```

```
        default:
            if(chk_char(in_str[char_index])) {
                return(tempresult);
            }
            else {
                if(char_index > 0 && got_first)
                    char_index--;
                while(char_index <= strlen(in_str)) {
                    if(chk_char(in_str[char_index]))
                        break;
                    else
                        *bad_input++ = in_str[char_index];
                    char_index++;
                }
                *bad_input = '\0';
                printf("\t>>> bad input: %s\n", bad_str);
                char_index++;
                return(INPUTERROR);
            } } }
    return(INPUTDONE);
}

BOOL chk_char(ch)
char ch;
{
    if(isspace(ch) || ch == '\0')
        return(OK);
    else
        return(NOT_OK);
}
```

FIGURE 92 Sample "C" Program

17.2 Preprocess, Instrument, Compile and Link

The first stage in *S-TCAT/C* is to prepare your "C" program to provide call-pair coverage data. Follow these steps:

1. Preprocessing the program. Most "C" compilers have this facility.
2. Instrumenting the program to insert markers at every segment position.

The following program shows, in bold, the effects of *S-TCAT/C* instrumentation on your "C" program:

```

/*
-----
-- S1 instrumentation by S-TCAT/C instrumenter:

-- Program s-ic, Release 8

-- Instrumented on Wed Jul  3 15:23:28 1991

-- SR Copy Identification No. 0.
-----
-- (c) Copyright 1991 by Software Research, Inc. All Rights
Reserved.
--
-- This program was instrumented by SR proprietary software,
-- for use with the SR proprietary S-TCAT runtime package.
-- Use of this program is limited by associated software
-- license agreements.
-----
*/

extern Strace();
extern Ftrace();
extern EntrMod();
extern ExtMod();
extern TCATFH();

char menu[13][79] = {
    "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
    "\n",
    "    What type of food would you like?\n",
    "\n",
    "    1        American 50s   \n",
    "    2        Chinese       - Hunan $tyle \n",

```



```
        "      3      Chinese   - Seafood Oriented \n",
        "      4      Chinese   - Conventional Style \n",
        "      5      Danish     \n",
        "      6      French     \n",
        "      7      Italian    \n",
        "      8      Japanese   \n",
        "\n\n"
};

int char_index;

main(argc,argv)
int      argc;
char     *argv[];
{
    int i, choice, c,answer;
    char str[79];
    int ask, repeat;
    int proc_input();

    Strace("IC",0x7504,0,0);
    EntrMod(-1,"main",3);

    c = 3;
    repeat = 1;

    while(repeat) {
        printf("\n\n\n");
        for(i = 0; i < 13; i++)
            printf("%s", menu[i]);

        (TCATFH(1),/*gets*/ gets(str)  ) ;
        printf("\n");
        while(choice = (TCATFH(2),/*proc_input*/ proc_input(str)
)
            {
                switch(choice) {
                    case 1:
                        printf("\tFog City Diner 1300 Battery 982-2000
\n");
                        break;
                    for(ask = TRUE; ask; ) {
                        case 2:
                            printf("\tHunan Village Restaurant 839 Kearney
956-7868 \n");
                            break;
                        case 3:
                            printf("\tOcean Restaurant726 Clement 221-3351
\n");
```

```

        break;
    case 4:
        printf("\tYet Wah 1829 Clement 387-8056 \n");
        break;
    case 5:
        printf("\tEiners Danish Rest. 1901 Clement386-9860
\n");

        break;
    case 6:
        printf("\tChateau Suzanne 1449 Lombard 771-9326
\n");

        break;
    case 7:
        printf("\tGrifone Ristorante1609 Powell 397-8458
\n");

        break;
    case 8:
        printf("\tFlints Barbecue4450 Shattuck, Oakland
\n");

        break;
    default:
        if(choice != -1)
            printf("\t>>> %d: not a valid choice.\n",
choice);

        break;
    }
}
for(ask = 1; ask; ) {
    printf("\n\tDo you want to run it again? ");
    while((answer = (--((&iob[0]))->_cnt < 0 ?
        (TCATFH(3),/*_filbuf*/ _filbuf((&iob[0])) )
        : (int) *((&iob[0]))->_ptr++)) != '\n') {
        switch(answer) {
            case 'Y':
            case 'y':
                ask = 0;
                char_index = 0;
                break;
            case 'N':
            case 'n':
                ask = 0;
                repeat = 0;
                break;
            default:
                break;
        }
    }
}
}
ExtMod("main");

```

```

        Ftrace(0);
    }

    int proc_input(in_str)
    char *in_str;
    {
        int tempresult = 0;
        char bad_str[80], *bad_input;
        int got_first = 0;
        EntrMod(-1, "proc_input", 5);

        bad_input = bad_str;

        while(
            (TCATFH(1), /*isspace*/ isspace(in_str[char_index]) ) )
            char_index++;
        for(ask = TRUE; ask; ) {
            for( ; char_index <= (TCATFH(2), /*strlen*/ strlen(in_str)
); char_index++) {
                switch(in_str[char_index]) {
                    case '0':
                    case '1':
                    case '2':
                    case '3':
                    case '4':
                    case '5':
                    case '6':
                    case '7':
                    case '8':
                    case '9':

                        tempresult = tempresult * 10 + (in_str[char_index] -
'0');

                        got_first = 1;
                        break;

                    default:
                        if( (TCATFH(3), /*chk_char*/ chk_char(in_str[char_in-
dex]) ) ) {
                            {ExtMod("proc_input");
                                return(tempresult); }
                        }
                    else {
                        if(char_index > 0 && got_first)
                            char_index--;
                        while(char_index <= (TCATFH(4), /*strlen*/ strlen(-
in_str) ) )
                            {
                                if( (TCATFH(5), /*chk_char*/ chk_char(in_str[-
char_index]) ) ) )

```

```
        break;
    else
        *bad_input++ = in_str[char_index];
        char_index++;
    }
    *bad_input = '\0';
    printf("\t>>> bad input: %s\n", bad_str);
    char_index++;

    { ExtMod("proc_input");
      return(-1); }
    } }
}

ExtMod("proc_input");
return(0); }

ExtMod("proc_input");
}

int chk_char(ch)
char ch;
{
    EntrMod(-1, "chk_char", 1);

    if( (TCATFH(1), /*isspace*/ isspace(ch) ) || ch == '\0' )
        { ExtMod("chk_char");
          return(1); }
    else
        {ExtMod("chk_char");
          return(0); }

    ExtMod("chk_char");
}
}
```

FIGURE 93 Instrumented Program Segment

17.3 Reference Listing

The **Reference Listing** file (that is *filename.i.A* or *filename.ia* for DOS) is produced by the instrumentor and is used for manual cross-referencing during a series of tests. The **Reference Listing** is a version of your "C" program which has a call-pair (or function call) marked.

You will use this report by gathering the **Not Hit** call-pair from a **Not Hit** report, and then looking up the related code in the **Reference Listing**. After reviewing the exercised and not-exercised parts of the program, you can design subsequent test cases to exercise more call-pairs.

Extensive call-pair and module notation have also been embedded and the call-pair sequence numbers are listed along the leftmost column.

The header identifies the file as a **Reference Listing** and includes the Release number plus a copyright notice. The code that **s-ic** adds appears in bold in the following program.

```
-----
-- S-TCAT/C, Release 8

-- (c) Copyright 1991 by Software Research, Inc. ALL RIGHTS
RESERVED.
-- CALL PAIR REFERENCE LISTING

-- Instrumentation date: Wed Jul  3 15:23:28 1991

-- Separate modules and call pair definitions for each module are
-- indicated in this commented version of the supplied source file.
-----
```

```
char menu[13][79] = {
    "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
    "\n",
    "    What type of food would you like?\n",
    "\n",
    "    1      American 50s  \n",
    "    2      Chinese  - Hunan Style \n",
    "    3      Chinese  - Seafood Oriented \n",
    "    4      Chinese  - Conventional Style \n",
    "    5      Danish      \n",
    "    6      French      \n",
    "    7      Italian     \n",
    "    8      Japanese    \n",
    "\n\n"
};
int char_index;
main(argc,argv)
```

```

int     argc;
char    *argv[];
{
    int  i, choice, c, answer;
    char str[79];
    int  ask, repeat;

/** Module main **/

    int  proc_input();
        c = 3;
        repeat = 1;
        while(repeat) {
            printf("\n\n\n");
            for(i = 0; i < 13; i++)
                printf("%s", menu[i]);
            gets(str);

/** Call-pair 1 **/
            printf("\n");
            while(choice = proc_input(str)) {

/** Call-pair 2 **/
                switch(choice) {
                    case 1:
                        printf("\tFog City Diner 1300 Battery982-2000 \n");
                        break;
                    case 2:
                        printf("\tHunan Village Restaurant
839 Kearney    956-7868 \n");
                        break;
                    case 3:
                        printf("\tOcean Restaurant 726 Clement
221-3351 \n");
                        break;
                    case 4:
                        printf("\tYet Wah 1829 Clement    387-
8056 \n");
                        break;
                    case 5:
                        printf("\tEiners Danish Restaurant 1901
Clement    386-9860 \n");
                        break;
                    case 6:
                        printf("\tChateau Suzanne 1449 Lombard
771-9326 \n");
                        break;
                    case 7:
                        printf("\tGrifone Ristorante 1609 Pow-
ell    397-8458 \n");
                        break;

```

```

        for(ask = TRUE; ask; ) {
            case 8:
                printf("\tFlints Barbecue 4450 Shat-
tuck, Oakland \n");
                break;
            default:
                if(choice != -1)
                    printf("\t>>> %d: not a valid
choice.\n", choice);
                break;
        }
    }
    for(ask = 1; ask; ) {
        printf("\n\tDo you want to run it again? ");
        while((answer = (--((&_iob[0]))->_cnt < 0 ?
_filbuf((&_iob[0])) : (int) *((&_iob[0]))->_ptr++)) != '\n') {
            /** Call-pair 3 **/

            switch(answer) {
                case 'Y':
                case 'y':
                    ask = 0;
                    char_index = 0;
                    break;
                case 'N':
                case 'n':
                    ask = 0;
                    repeat = 0;
                    break;
                default:
                    break;
            } } } }
    int proc_input(in_str)
    char *in_str;
    {
        int tempresult = 0;
        char bad_str[80], *bad_input;

        /** Module proc_input **/

        int got_first = 0;
        bad_input = bad_str;
        while(isspace(in_str[char_index]))
            /** Call-pair 1 **/
                char_index++;
        for( ; char_index <= strlen(in_str); char_index++)
            {
                /** Call-pair 2 **/
                switch(in_str[char_index]) {

```

```

        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':

        tempresult = tempresult * 10 + (in_str[
char_index] - '0');
        got_first = 1;
        break;
default:
    if(chk_char(in_str[char_index])) {
/** Call-pair 3 **/
        return(tempresult);
    }
    else {
        if(char_index > 0 && got_first)
            char_index--;
        for(ask = TRUE; ask; ) {
/** Call-pair 4 **/
            while(char_index <= strlen(in_str)) {
/** Call-pair 5 **/
                if(chk_char(in_str[char_index]))
                    break;
                else
                    *bad_input++ = in_str[char_in-
dex];
                char_index++;
            }
            *bad_input = '\0';
            printf("\t>>> bad input: %s\n", bad_
str);
            char_index++;
            return(-1);
        } }
    return(0);
}
int chk_char(ch)
char ch;

/** Module chk_char **/
{

```



```
        if(isspace(ch) || ch == '\0')
/** Call-pair 1 **/
        return(1);
        else
        return(0);
    }
-----
-- S-TCAT/C, Release 8
-- END OF S-TCAT/C CALL PAIR REFERENCE LISTING
-----
```

FIGURE 94 Reference Listing

17.4 Instrumentation Statistics

The instrumentor also produces program statistics. They are organized module-by-module.

```
-----  
-- S-TCAT/C, Release 8.  
  
-- (c) Copyright 1991 by Software Research, Inc.  ALL RIGHTS RESERVED.  
-- INSTRUMENTATION STATISTICS  
-- Instrumentation date: Wed Jul 3 15:23:28 1991  
-----  
MODULE 'main':  
  
statements = 42  
compound statements = 7  
  
call pairs found = 16  
call pairs instrumented = 16  
  
return statement = 0  
  
MODULE 'proc_input':  
  
statements = 22  
compound statements = 6  
  
call pairs found = 6  
call pairs instrumented = 6  
  
return statements = 3  
  
MODULE 'chk_char':  
  
statements = 2  
compound statement = 1  
  
call pair found = 1  
call pair instrumented = 1  
  
return statements = 2  
  
-----  
-- S-TCAT/C.  
  
-- END OF S-TCAT/C INSTRUMENTATION STATISTICS  
-----
```

FIGURE 95 Instrumentation Statistics Sample

17.5 Call-Pair Listing

The Call-Pair Listing file (that is *filename.i.P* or *filename.ip* for DOS) is produced by the instrumentor. It is used by the **Xcalltree** utility. It lists all the call-pairs encountered in the *filename.c* file.

Below is the call-pair listing file for the *example.c* program.

```
main    printf
main    printf
main    gets
main    printf
main    proc_input
main    printf
main    printf
main    printf
main    printf
main    printf
main    printf
main    printf
main    printf
main    printf
main    printf
main    _filbuf
proc_inputstrlen
proc_inputchk_char
proc_inputstrlen
proc_inputchk_char
proc_inputprintf
```

FIGURE 96 Call-Pair Listing Example

17.6 Reading S-TCAT Reports

The last and most important step in test analysis is to obtain test coverage analysis reports. This section details how to read reports: the **Cumulative, Past, Not Hit, Hit, Newly Hit, Newly Missed, Linear Histogram** and **Logarithmic Histogram** reports. These reports analyze the trace file and archive data produced and present it in an easy-to-read format. Of particular importance are the **Cumulative** and **Not Hit** reports. To obtain these reports, use the following command and, if necessary, include the trace file name.

```
scover [tracefile name] -c -n
```

This produces a file called *Coverage* which contains **Cumulative** and **Not Hit** reports plus an archive file, *Archive*, which contains coverage data accumulated to this point and can be used in later testing. View *Coverage* to see your reports with any non-document editor such as **VI** or **Word**. The following subsections describe each coverage report in detail.

17.6.1 Cumulative Report

As shown in the following figure, the **Cumulative Report** lists each module by name and indicates the number of call-pairs. The report tells you how many times each module was invoked, how many of its call-pairs were hit, and its resulting S1 coverage measure. For instance, module `porc_input` might have 5 call-pairs, or function calls. If 3 of them were exercised, the S1 metric for that module would be $3 / 5 = 60.00\%$. For the S1 metric, a hit call-pair is counted only once, regardless of how many times it was actually hit. The report tells about the current and all tests, including previous tests. Current test data comes from the latest trace file - the cumulative summary information from the archive file. In addition, any text you entered earlier as a trace descriptor is shown at the bottom of the report.

```
Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

```
Selected SCOVER System Option Settings:
```

```
[-c] Cumulative Report    -- YES
[-p] Past Report         -- NO
[-n] Not Hit Report      -- NO
[-H] Hit Report          -- NO
[-nh] Newly Hit Report   -- NO
[-nm] Newly Missed Report -- NO
[-h] Histogram Report    -- NO
[-l] Log Scale Histogram -- NO
[-Z] Reference Listing S1 -- NO
```

```

Options read:  1
S-TCAT: Coverage Analyzer.  [Release 8]
(c) Copyright 1991 by Software Research, Inc.  ALL RIGHTS RESERVED.
-----+-----
-+
|                                     | Current Test          | Cumulative Summary    |
|                                     |-----+-----|-----+-----|
|                                     | No. Of               | No. Of               |
| Module      Number Of | No. Of Call-pairs  S1% | No. Of Call-pairs  S1% |
| Name:       Call-pairs: | Invokes  Hit    Cover | Invokes  Hit    Cover |
|-----+-----|-----+-----|-----+-----|
-+
| main          3 | 1   3 100.00 | 1   3 100.00 |
| proc_input   5 | 2   3  60.00 | 2   3  60.00 |
| chk_char     1 | 2   1 100.00 | 2   1 100.00 |
|-----+-----|-----+-----|
-+
| Totals       9 | 5   7  77.78 | 5   7  77.78 |
|-----+-----|-----+-----|
-+

Current test message(s) (saved in archive):
Runtime vers 4.9, last updated 7/31/89

```

FIGURE 97

Cumulative Coverage Report

The cumulative coverage report contains the following information:

1. "Module Name" lists the names of each module in the program
2. "Number of Call-pairs" lists the number of call-pairs in each module.
3. "No. of Invokes" for "Test" gives the number of times the module was called during the test run.
4. "No. of Call-pairs Hit" for "Test" indicates how many of the module's total call-pairs were exercised during the test run.
5. "S1% Cover" for "Test" provides a S1 value for each module for the current test.
6. "Cumulative Summary" refers to the second part of the report. This provides data for testing to date, including any archived data that has been submitted as input to scover with the -a option or the default Archive file. In the run for this example no archive data was included, so data for "Cumulative Summary" is the same as for "Test".
7. "Totals" shows the total for each category of test coverage data. This gives you immediate feedback on your program as a whole.
8. "Current Test Message" displays the trace file descriptor text typed in earlier.

17.6.2 Past Report

The **Past Report** (`-p` option) is similar in appearance to the standard coverage report, but it analyzes only one set of data: an archive file. The report summarizes the percentage of call-pairs hit in each module listed, giving the *S1* value for each module and the program as a whole.

Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

Selected SCOVER System Option Settings:

```
[-c] Cumulative Report      -- NO
[-p] Past Report           -- YES
[-n] Not Hit Report        -- NO
[-H] Hit Report            -- NO
[-nh] Newly Hit Report     -- NO
[-nm] Newly Missed Report  -- NO
[-h] Histogram Report     -- NO
[-l] Log Scale Histogram  -- NO
[-Z] Reference Listing S1  -- NO
```

Options read: 1

S-TCAT: Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

```
+-----+
|                                     |
|                                     | (Archived) Past Tests |
|                                     |
|-----+-----+
|                                     |
|                                     | Number Of | | | |
| Module                            | Number Of | Call-pairs | Percent |
| No. Name                           | Call-pairs: | Invocations | Hit    | Coverage |
|-----+-----+-----+-----+
| 0: main                             | 3 |         | 1      | 3     | 100.00 |
| 1: proc_input                       | 5 |         | 2      | 3     | 60.00  |
| 2: chk_char                         | 1 |         | 2      | 1     | 100.00 |
|-----+-----+-----+-----+
| Totals                             | 9 |         | 5      | 7     | 77.78  |
|-----+-----+-----+-----+
```

Current test message(s) (saved in archive):

Runtime vers 4.9, last updated 7/31/89

17.6.3 Not Hit Report

The **Not Hit Report** (`-n` option), illustrated below, analyzes your program from an analytical perspective, showing which call-pairs were not hit. You are given the module's name and the identification number for each call-pair not hit in the current test. To identify the actual code not executed and plan new test cases, look up the in the **Reference Listing**. For *S-TCAT/C*, this is the file *filename.i.A* (for UNIX) or *filename.iA* (for DOS).

Occasionally, all call-pairs in a module are hit during a test. When this happens, a special message is displayed. However, in the example, each module had call-pairs that were not hit. In most cases, at least one segment in each module has not been hit. This report ends with a short summary of test results, including the number of call-pairs hit in the instrumented program, the total number of call-pairs in the instrumented program, and the S1 coverage value for this test.

Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

Selected SCOVER System Option Settings:

```
[-c] Cumulative Report    -- NO
[-p] Past Report          -- NO
[-n] Not Hit Report       -- YES
[-H] Hit Report           -- NO
[-nh] Newly Hit Report    -- NO
[-nm] Newly Missed Report -- NO
[-h] Histogram Report     -- NO
[-l] Log Scale Histogram  -- NO
[-Z] Reference Listing S1 -- NO
```

Options read: 1
S-TCAT: Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

S1 Not Hit Report.

No.	Module Name:	Call-pair Coverage Status:
1	main	All Call-pairs Hit. S1 = 100%
2	proc_input	4 5
3	chk_char	All Call-pairs Hit. S1 = 100%

Number of Call-pairs Not Hit: 2
Total Number of Call-pairs: 9
S1 Coverage Value: 77.78%

FIGURE 98 Not Hit Report

17.6.4 Hit Report

The **Hit Report** (**-H** option) identifies all of the call-pairs which were exercised in the present and past tests. It analyzes both the trace file and archive files.

Here is a sample of the **Hit Report**.

```
Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

Selected SCOVER System Option Settings:

```
[-c] Cumulative Report    -- NO
[-p] Past Report          -- NO
[-n] Not Hit Report       -- NO
[-H] Hit Report           -- YES
[-nh] Newly Hit Report    -- NO
[-nm] Newly Missed Report -- NO
[-h] Histogram Report     -- NO
[-l] Log Scale Histogram  -- NO
[-Z] Reference Listing S1 -- NO
```

Options read: 1

```
S-TCAT: Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

S1 Call-pair Hit Report.

No.	Module Name:	Call-pair Coverage Status:
1	main	All Call-pairs Hit. S1 = 100%
2	proc_input	1 2 3
3	chk_char	All Call-pairs Hit. S1 = 100%
Number of Call-pairs Hit:		7
Total Number of Call-pairs:		9
S1 Coverage Value:		77.78%

FIGURE 99 Hit Report

17.6.5 Newly Hit Report

The **Newly Hit Report** (**-NH** option) identifies all call-pairs that are hit in the present test but which were not hit in any prior test. Here is a sample of the **Newly Hit Report**.

```
Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

```
Selected SCOVER System Option Settings:
```

```
[-c] Cumulative Report    -- NO
[-p] Past Report          -- NO
[-n] Not Hit Report       -- NO
[-H] Hit Report           -- NO
[-nh] Newly Hit Report    -- YES
[-nm] Newly Missed Report -- NO
[-h] Histogram Report     -- NO
[-l] Log Scale Histogram  -- NO
[-Z] Reference Listing S1 -- NO
```

```
Options read: 1
S-TCAT: Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

```
S1 Call-pair Newly Hit Report.
```

```
No.    Module Name:          Call-pair Coverage Status:
2      proc_input
                               4    5
```

FIGURE 100 Newly Hit Report**17.6.6 Newly Missed Report**

This report (**-NM** option) displays what the current test lost.

```
Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

```
Selected SCOVER System Option Settings:
```

```
[-c] Cumulative Report    -- NO
[-p] Past Report          -- NO
[-n] Not Hit Report       -- NO
[-H] Hit Report           -- NO
[-nh] Newly Hit Report    -- NO
[-nm] Newly Missed Report -- YES
```

```

[-h] Histogram Report      -- NO
[-l] Log Scale Histogram  -- NO
[-Z] Reference Listing S1 -- NO

```

Options read: 1

S-TCAT: Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

S1 Call-pair Newly Missed Report.

```

No.      Module Name:      Call-pair Coverage Status:
2        proc_input
                                     4    5

```

FIGURE 101 Newly Missed Report

17.6.7 Linear Histogram

This report (**-h** option) displays a mark for each time a call-pair is hit during testing. The samples shown are for the modules *main* and *proc_input*.

Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

Selected SCOVER System Option Settings:

```

[-c] Cumulative Report      -- NO
[-p] Past Report           -- NO
[-n] Not Hit Report         -- NO
[-H] Hit Report            -- NO
[-nh] Newly Hit Report     -- NO
[-nm] Newly Missed Report  -- NO
[-h] Histogram Report      -- YES
[-l] Log Scale Histogram   -- NO
[-Z] Reference Listing S1  -- NO

```

Options read: 1

S-TCAT: Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

Call-pair Level Histogram for Module: main

```

                                     +-----+
                                     | Number of Executions, Normalized to Maximum
                                     | (Maximum =      13 Hits)  X = One Hit
                                     | (Scale:      7.692      Each X =    0.260 Hits)
Call-pair Number of |
Number Executions >-1-----20-----40-----60-----80-----100
+-----+
|

```

CHAPTER 17: Full S-TCAT Example

```
1          7 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
2         13 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
3          7 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          |
+-----+

Average Hits per Executed Call-pair:  9.0000
S1 Value for this Module:             100.0000

S-TCAT: Coverage Analyzer.  [Release 8]
(c) Copyright 1991 by Software Research, Inc.  ALL RIGHTS RESERVED.
Call-pair Level Histogram for Module: proc_input

          +-----+
          | Number of Executions, Normalized to Maximum
          | (Maximum =    18 Hits)  X = One Hit
          | (Scale:    5.556      Each X =    0.360 Hits)
Call-pair Number Of |
Number   Executions >-1-----20-----40-----60-----80-----100
+-----+
          |
1         14 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
2         18 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
3         12 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
4          5 | XXXXXXXXXXXXXXXX
5          5 | XXXXXXXXXXXXXXXX
          |
+-----+

Average Hits per Executed Call-pair:  10.8000
S1 Value for this Module:             100.0000
```

FIGURE 102 Linear Histogram

17.6.8 Logarithmic Histogram

This report (`-l` option) is similar to the linear histogram but translates the data into logarithms to make the report more readable when some call-pairs have been hit many times and others fewer times. The samples are for the modules *main* and *proc_input*.

Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

Selected SCOVER System Option Settings:

```
[-c] Cumulative Report      -- NO
[-p] Past Report           -- NO
[-n] Not Hit Report        -- NO
[-H] Hit Report            -- NO
[-nh] Newly Hit Report     -- NO
[-nm] Newly Missed Report  -- NO
[-h] Histogram Report     -- NO
[-l] Log Scale Histogram  -- YES
[-Z] Reference Listing S1 -- NO
```

Options read: 1

S-TCAT: Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

Call-pair Level Histogram for Module: main

```

+-----+
| Logarithm of Executions, Normalized to Maximum
| (Maximum =      14 Hits)
|
Call-pair Number Of |
Number   Executions >-----1-----10-----20----30---40--80-100
+-----+
|
| 1           8 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
| 2          14 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
| 3           8 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|
+-----+

```

Average Hits per Executed Call-pair: 10.0000

S1 Value for this Module: 100.0000

S-TCAT: Coverage Analyzer. [Release 8]

(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.

Call-pair Level Histogram for Module: proc_input

```

+-----+
| Logarithm of Executions, Normalized to Maximum
| (Maximum =      19 Hits)
|
Call-pair Number Of |

```

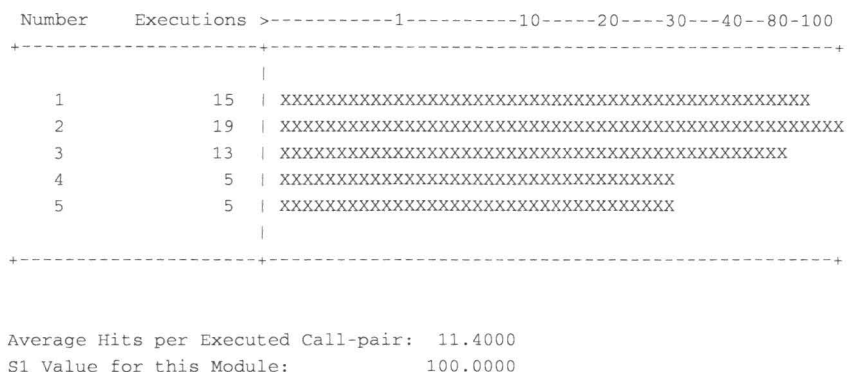


FIGURE 103 Logarithmic Histogram

17.6.9 Reference Listing S1 Report

This report (**-z** option) analyzes the specified reference listing file and produces a report that shows the coverage level achieved for all modules that are named in the specified reference listing. The following figure shows a partial reference listing of the *example.c* program. Statistics of coverage appear in bold on the left-hand column.

```
Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

Selected SCOVER System Option Settings:

```

[-c] Cumulative Report      -- NO
[-p] Past Report            -- NO
[-n] Not Hit Report         -- NO
[-H] Hit Report             -- NO
[-nh] Newly Hit Report      -- NO
[-nm] Newly Missed Report   -- NO
[-h] Histogram Report       -- NO
[-l] Log Scale Histogram    -- NO
[-Z] Reference Listing S1   -- YES

```

```
Options read:  1
S-TCAT: Coverage Analyzer. [Release 8]
(c) Copyright 1991 by Software Research, Inc. ALL RIGHTS RESERVED.
```

S-TCAT Coverage on Reference Listing Report, based on file *example.i.A*.

(Coverage values for all tests processed are reported in left-hand column.
 "*****" indicates not hits on corresponding call-pair. Extra names not
 part of this listing but in the Archive file are ignored.)

-- (c) Copyright 1991 by Software Research, Inc. ALL RIGHTS
 RESERVED.

-- CALL PAIR REFERENCE LISTING

```

char menu[13][79] = {
    "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
    "\n",
    "    What type of food would you like?\n",
    "    \n",
    "    1   American 50s   \n",
    "    2   Chinese     - Hunan Style \n",
    "    3   Chinese     - Seafood Oriented \n",
    "    4   Chinese     - Conventional Style \n",
    "    5           Danish           \n",
    "    6           French           \n",
    "    7           Italian          \n",
    "    8           Japanese        \n",
    "\n\n"
};
int char_index;
main(argc,argv)
int  argc;
char *argv[];
{
    int i, choice, c,answer;
    char str[79];
    int ask, repeat;

```

S1 = 100.00 /** Module main **/

```

int proc_input();
c = 3;
repeat = 1;
while(repeat) {
    printf("\n\n\n");
    for(i = 0; i < 13; i++)

```

```

        printf("%s", menu[i]);
        gets(str);
1      /** Call-pair 1 **/
        printf("\n");

        while(choice = proc_input(str)) {
2      /** Call-pair 2 **/
            switch(choice) {
                case 1:
                    printf("\tFog City
Diner      1300 Battery    982-2000 \n");
                    break;
                case 2:
                    printf("\tHunan Village Restau-
rant 839 Kearney    956-7868 \n");
                    break;
                case 3:
                    printf("\tOcean Restau-
rant      726 Clement    221-3351 \n");
                    break;
                case 4:
                    printf("\tYet
Wah      1829 Clement    387-8056 \n");
                    break;
                case 5:
                    printf("\tEiners Danish Restau-
rant 1901 Clement    386-9860 \n");
                    break;
                case 6:
                    printf("\tChateau
Suzanne    1449 Lombard    771-9326 \n");
                    break;
                case 7:
                    printf("\tGrifone Ris-
torante    1609 Powell    397-8458 \n");
                    break;
                case 8:
                    printf("\tFlints Barbe-
cue      4450 Shattuck, Oakland \n");
                    break;
                default:
                    if(choice != -1)
                        printf("\t>>> %d: not a valid
choice.\n", choice);
                    break;
            }
        }
        for(ask = 1; ask; ) {
            printf("\n\tDo you want to run it
again? ");

```

```

                                while((answer = (--(&_iob[0]))->_cnt
< 0 ? _filbuf((&_iob[0])) : (int) *((&_iob[0]))->_ptr++) != '\n') {
1                                /** Call-pair 3 */
                                    switch(answer) {
                                        case 'Y':
                                        case 'y':
                                            ask = 0;
                                            char_index = 0;
                                            break;
                                        case 'N':
                                        case 'n':
                                            ask = 0;
                                            repeat = 0;
                                            break;
                                        default:
                                            break;
                                    } } } } }

                                .
                                .
                                .

                                int chk_char(ch)
                                char ch;
S1 = 100.00    /** Module chk_char */

                                {
                                    if(isspace(ch) || ch == '\0')
2                                /** Call-pair 1 */
                                        return(1);
                                    else
                                        return(0);
                                }

```

FIGURE 104 Reference Listing S1 Report
17.7 **Summary**

After reviewing these reports (particularly the **Cumulative Report** and the **Not Hit Report**), you will typically rerun the tests with different or additional test cases, designed to exercise previously not-hit call-pairs and achieve a higher S1 value. The higher the S1 value, the more complete your testing. When you achieve a satisfactory value for S1, for example, 95 percent or more, you can stop testing.



Graphical User Interface (GUI) Tutorial

This chapter demonstrates using *S-TCAT* in the OSF/Motif environment.

18.1 Invocation

To invoke, type:

```
xstcat
```

The result is the main menu (shown below). This window has a window menu button (available for all windows) that allows the user to restore, move, size, minimize, lower and close the window. This menu button can be used at any time during the X Window System program. For closing main application windows, however, it is best to use the **System** menu's **Exit** option to prevent any system crashes. The two buttons in the upper right hand corner of the window allow the user to maximize or minimize the window size.

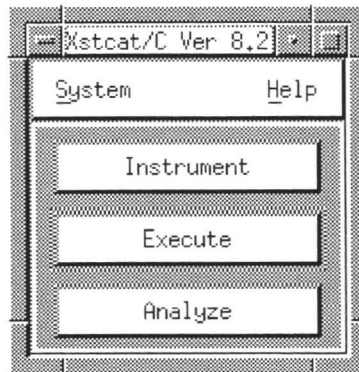


FIGURE 105 Main Menu

To invoke with **STW/COV**, click first on **Coverage** and then on **S-TCAT**.
The **S-TCAT** main menu pops up.

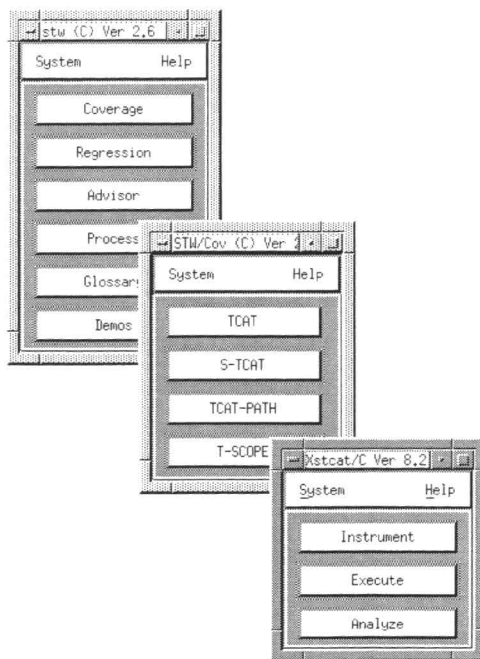


FIGURE 106 STW/COV Invocation
18.2 Using S-TCAT/C

For first time use, always read the help menus. Below is main menu's help, explaining *S-TCAT* three stages of testing: instrument, execute and analyze.

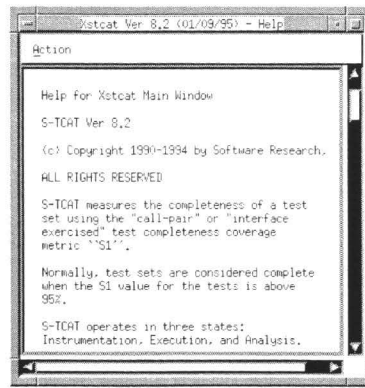


FIGURE 107 Main Menu Help

18.2.1 Instrument

S-TCAT instruments the source code of the program to be tested; that is, it inserts function calls at each call-pair. Double-click on **Instrument** in order to begin testing. There are a variety of options which can be selected with the menu in Figure 108 (see next page):

1. **Preprocessing** can be turned on or off. If it is turned off, then the instrumentor will not preprocess.
2. **Preprocessor output suffix** is set to `.i`, which is normally the extension for preprocessed "C" programs. This option is user editable.
3. **Preprocessor Command** is set to `cc -P`. Refer to Chapter 18 for further information. This option is user editable.
4. **Preprocessor options** are options in addition to the **Preprocessor command** previously specified.
5. **Instrumentor Command** is set to `s-i.c`. This option is user-editable.
6. **Instrumentor options**
 - **Recognize `_exit` as keyword** corresponds to the command line `-u` switch. Refer to Section 3.2.1.
 - **Do not recognize `exit` as keyword** corresponds to the command line `-x` switch. Refer to Section 3.2.1.

Do not instrument functions in file corresponds to the `-DI` switch. Specify a filename that contains lists of modules that are to be instrumented. Refer to Section 3.2.1.

Specify maximum file name length corresponds to the `-f1` value switch. Specify a number that will correspond to the maximum number of characters. Refer to Section 3.2.1.

Specify maximum function name length corresponds to the `fn` value switch. Specify a number that will correspond to the maximum number of characters. Refer to Section 3.2.1.

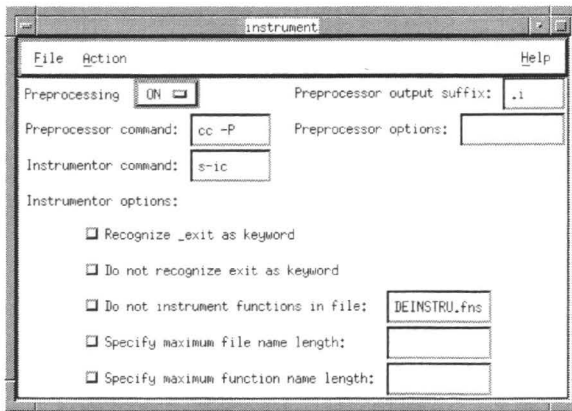


FIGURE 108 Instrument Menu

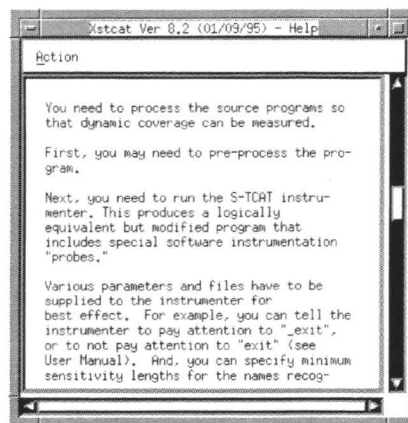


FIGURE 109 Instrument Help Menu

After selecting instrumentor options, do the following:

1. Make sure the **Preprocessing** switch is ON.
2. Click on the **File** pull-down menu. Drag the mouse down and select **Set File Name**. A file pop-up window appears (refer to the picture below). Select the file to be instrumented by either highlighting or typing it into the **Selection** box. Press **OK**.
3. After establishing the file to be instrumented, click on the **Action** pull-down menu. Drag the mouse down and select **Preprocess** and then **Instrument**. Note: **Instrument** cannot be selected until preprocessing has been completed.

NOTE: Current status and errors are displayed in the invocation box from time to time. Frequently refer to the box while testing to see where system crashes, errors and passes occur. When finished, click on **Exit** under the **File** pull-down menu.

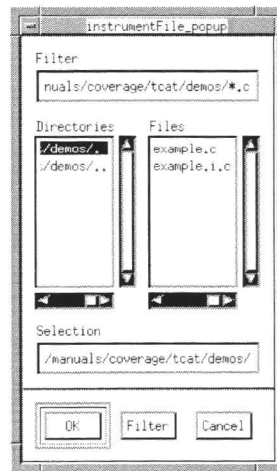


FIGURE 110 File Pop-Up Menu

18.2.2 Execute

The **Execute** menu compiles, links and executes the program. Normally, the user compiles the instrumented source file and then links all the

source files with the runtime object module (which is specified under the **File** pull-down menu). The user can also use the Make file. Both methods are explained in this section.

Double-click on **Execute** to begin. The menu below appears.

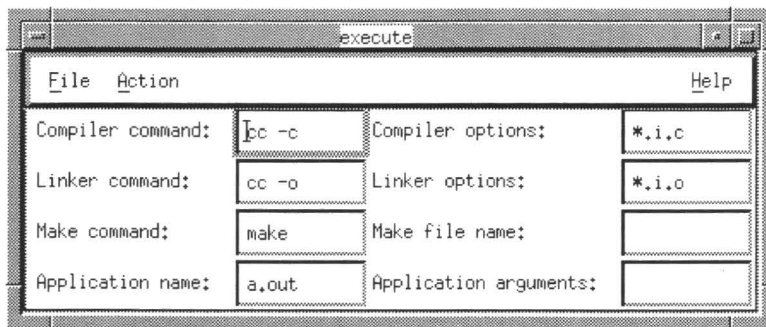


FIGURE 111

Execute Menu

There are a variety of options which can be selected from the **Execute** menu.

1. **Compiler command** is used to invoke the compiler on the system. It is set to `cc -c` but is user-editable.
2. **Compiler options** are the options for the compiler. It is set to `*.i.c` but is user-editable.
3. **Linker command** is used to invoke link. It is set to `cc -o` but is user-editable.
4. **Linker options** are the options used in order to link. It is set to `*.i.o` but is user-editable.
5. **Make command** is used to invoke the make utility.
6. **Make file name** is where the make file is specified. It is fixed to `Makefile` but is user-editable.
7. **Application name** is the command used to invoke the instrumented program. It is fixed to `a.out` but is user-editable.

8. **Application argument** is where command line arguments are specified. It is user-editable.

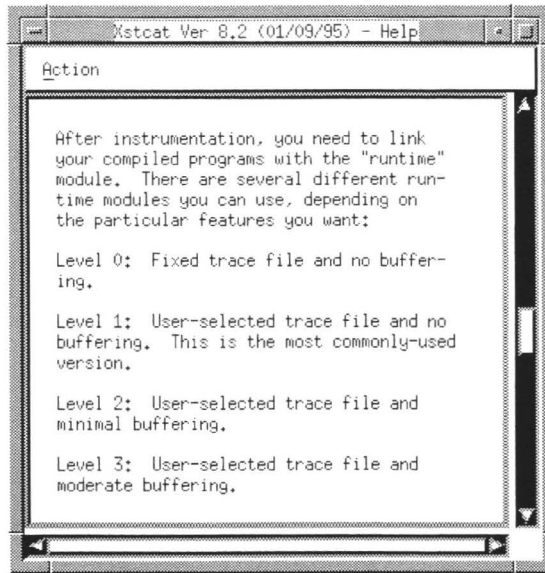


FIGURE 112 Execute Help Menu

Execute one of two ways:

1. Without Make File
 - (a) Click on the **File** pull-down menu, drag the mouse to **Set Runtime Object Module** and click. A pop-up window appears (shown in Figure 113). Highlight or type in (the Selection Box) the necessary file. Click **OK**. Refer to the help frame and to Section 12.1 and 12.2 for SR supplied runtime object modules.
 - (b) Set the compiler and linker commands (that is **Compiler command**, **Compiler options**, **Linker command** and **Linker options**) as appropriate.
 - (c) Click on the **Action** pull-down menu and select **Compile**. When completed, the invocation window will state so.
 - (d) Click on **Link**. Invocation window will indicate when linking has occurred.
 - (e) Click on **Run Application**.
2. With Make File: make organizes all compiler and linker commands and files.

- (a) Click on the **File** pull-down menu, drag the mouse to **Set Runtime Object Module** (shown in Figure 113) and click **Highlight** or type in (the **Selection** box) the necessary filename. Click **OK**. Refer to the help frame and to Section 12.1 and 12.2 for SR-supplied runtime object modules.
- (b) Set the make commands (that is **Make Command**, **Make file name**, **Application name** and **Application arguments**) as appropriate.
- (c) Click on the **Action** pull-down menu and select **Make**. When completed the invocation window will state so.
- (d) Click on **Run Application**.

Whichever method is chosen, the trace file is created.

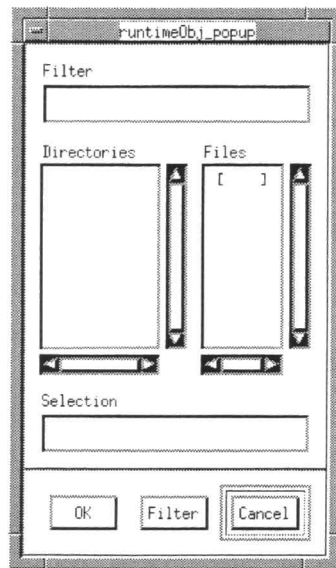


FIGURE 113 Runtime Object Module Pop-Up Screen

18.2.3 Analyze

After executing your program, you can analyze the trace file using the cover command. Double-click on **Analyze** and the menu below appears.

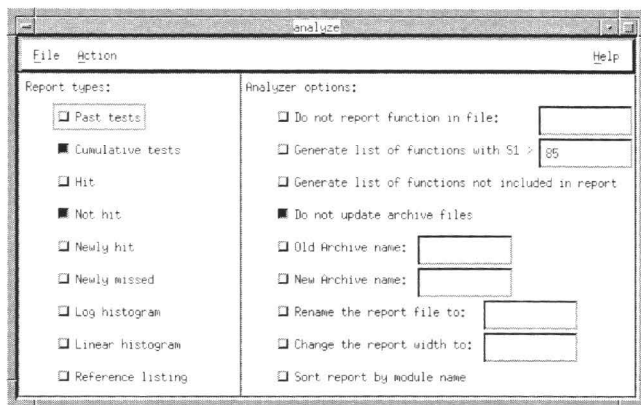


FIGURE 114 Analyze Menu

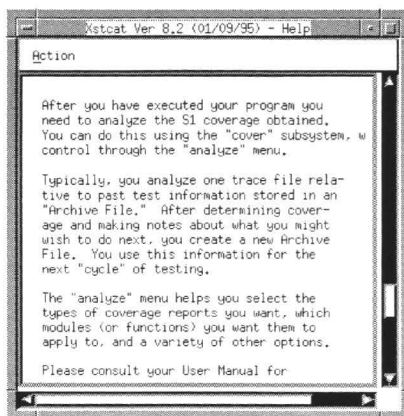


FIGURE 115 Analyze Help Menu

There are a variety of options which can be selected with the **Analyze** menu.

1. Coverage Reports:

- (a) **Past test** corresponds to the `-p` command line option. (Refer to sections 5.1.4 and 10.6.2 for further information.)
- (b) **Cumulative test** corresponds to the `-c` command line option. (Refer to sections 5.1.4 and 10.6.1 for further information.)
- (c) **Hit** corresponds to the `-H` command line option. (Refer to sections 5.1.4 and 10.6.4 for further information.)
- (d) **Not Hit** corresponds to the `-n` command line option. (Refer to sections 5.1.4 and 10.6.4 for further information.)
- (e) **Newly hit** corresponds to the `-NH` command line option. (Refer to sections 5.1.4 and 10.6.5 for further information.)
- (f) **Newly missed** corresponds to the `-NM` command line option. (Refer to sections 5.1.4 and 10.6.6 for further information.)
- (g) **Linear histogram** corresponds to the `-h` command line option. (Refer to sections 5.1.4 and 10.6.7 for further information.)
- (h) **Log histogram** corresponds to the `-l` command line option. (Refer to sections 5.1.4 and 10.6.8 for further information.)
- (i) **Reference listing** corresponds to the `-Z` command line option. (Refer to sections 5.1.4 and 10.6.9 for further information.)

2. Analyzer Options:

- (a) **Do not report functions in file** corresponds to the `-DI` `deinst-file` command line option. (Refer to Section 5.1.4 for further information). Specify the file in the supplied box.
- (b) **Generate list of functions with C1>** corresponds to the `-T [threshold]` command line option (refer to Section 5.1.4 for further information). Specify the coverage threshold percent in the form of a real or decimal number in the supplied box.

- (c) **Do not update archive files** corresponds to the `-su` command line option. (Refer to Section 13.1.4 for further information).
- (d) **Old Archive name** corresponds to the `-a old-archive` command line option. (Refer to Section 13.1.4 for further information). Specify the file in the supplied box.
- (e) **New Archive name** corresponds to the `-f new-archive` command line option. (Refer to Section 13.1.4 for further information). Specify the file in the supplied box.
- (f) **Rename the report file** to corresponds to the `-r` report command line option. (Refer to Section 13.1.4 for further information). Rename the file in the supplied box.
- (g) **Change the report width** to corresponds to the `-w` width command line option. (Refer to Section 13.1.4 for further information). Specify a decimal number in the supplied box.
- (h) **Sort report by module name** corresponds to the `-s` command line option. (Refer to Section 13.1.4 for further information).

Analyze in the following way:

1. Click on the **File** pull-down menu, drag the mouse to **Set Input Trace File Name**, and click. A trace file pop-up window appears (shown in Figure 116). Highlight or type in the file in the **Selection** box. Click **OK**.
2. Select the coverage reports and analyzer options. For the purpose of this demonstration, **Past Test**, **Cumulative Test**, **Hit**, **Linear Histogram**, and **Reference Listing** reports have been selected.
 - (a) When selecting the **Reference listing** option, a reference listing pop-up window appears (shown in Figure 117). Select the file and click **OK**.
3. Click on the **Action** pull-down menu and select **Run Coverage Analyzer**.
4. Click on the **Action** pull-down menu and select **View Report**. View the reports by using the menu's scroll bars. Figures 118 through 123 reflect viewed reports.
5. Click on the **Action** pull-down menu and select **View Source**. **View Source** associates a segment or node with its corresponding source code (refer to Chapters 23 and 24 for further information).
 - (a) Click on this option and a pop-up window appears (see Figure 124). Select a file and click **OK**. For this demon-

stration, the main module has been selected (see Figure 125).

(b) Source view by clicking on a call-pair.

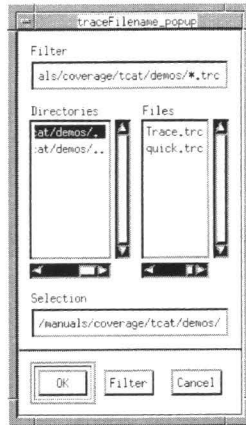


FIGURE 116 Set Input Trace File Name Pop-Up Window

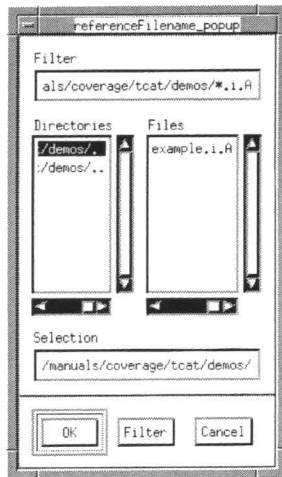


FIGURE 117 Reference Listing Pop-Up Window

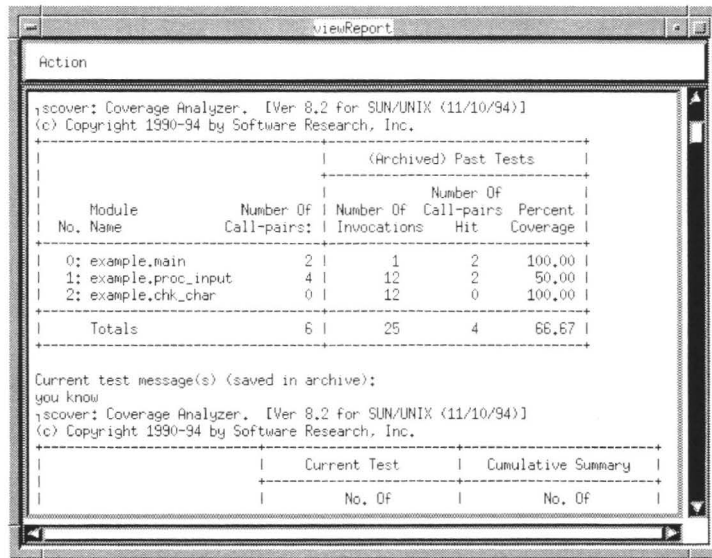


FIGURE 118 Past Test Report

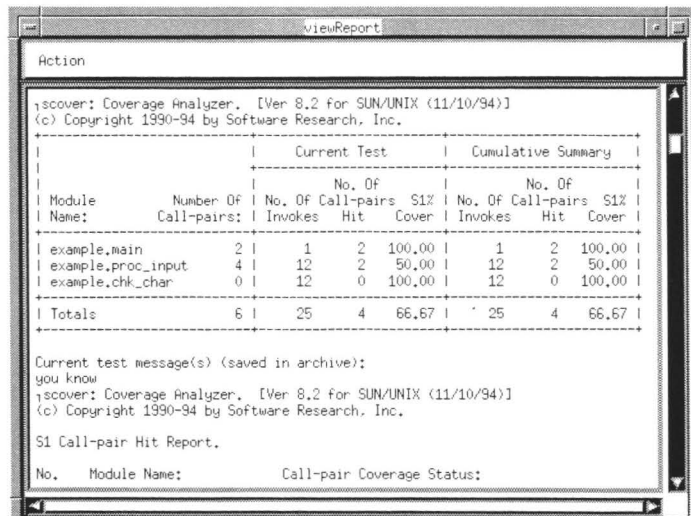


FIGURE 119 Cumulative Report

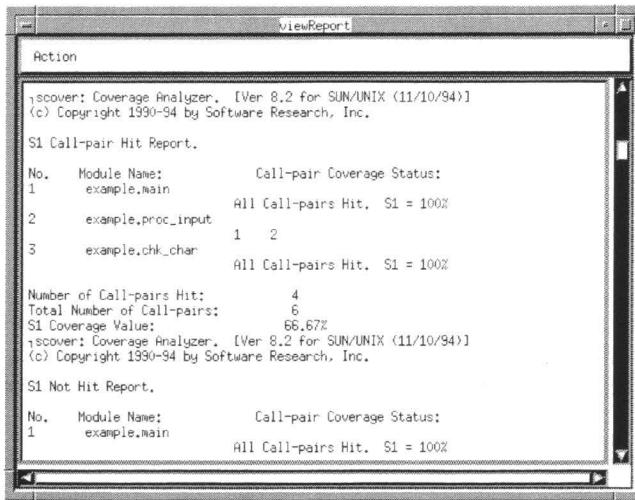


FIGURE 120 Hit Report

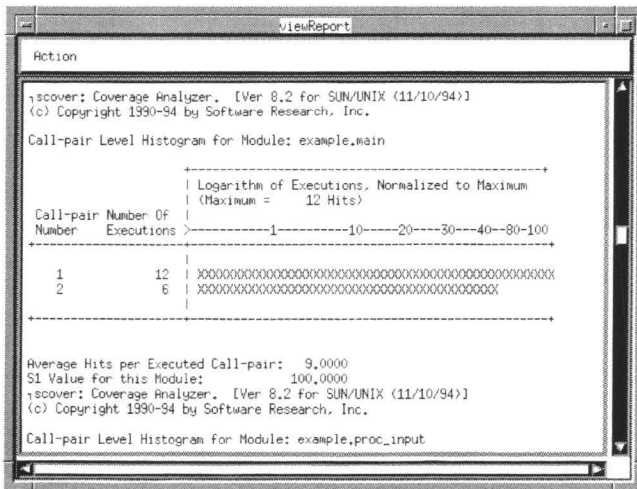


FIGURE 121 Linear Histogram

```

Action
-----
-- S-TCAT/C, Ver 8.2 For SUN (10/28/94).
--
-- (c) Copyright 1990 by Software Research, Inc. ALL RIGHTS RES
--
-- CALL PAIR REFERENCE LISTING
--
-- Instrumentation date: Tue Jun 20 15:08:52 1995
--
-- Separate modules and call pair definitions for each module are
-- indicated in this commented version of the supplied source f
-----
extern struct _iobuf {
    int _cnt;
    unsigned char *_ptr;
    unsigned char *_base;
    int _bufsiz;
    short _flag;
    char _file;
} _iobuf[];
extern struct _iobuf *_fopen();
extern struct _iobuf *_fdopen();
extern struct _iobuf *_freopen();

```

FIGURE 122 Reference Listing (Part 1 of 2)

```

Action
extern char *_ctermid();
extern char *_userid();
extern char *_tempnam();
extern char *_tmpnam();
extern char *_ctype[];
char menu[13][79] = {
    "SOFTWARE RESEARCH'S RESTAURANT GUIDE \n",
    "  What type of food would you like?\n",
    "\n",
    "  1      American 50s  \n",
    "  2      Chinese   - Hunan Style \n",
    "  3      Chinese   - Seafood Oriented \n",
    "  4      Chinese   - Conventional Style \n",
    "  5      Danish    \n",
    "  6      French    \n",
    "  7      Italian   \n",
    "  8      Japanese  \n",
    "\n\n"
};
int char_index;
main(argc,argv)
int  argc;
char *_argv[];
<

```

FIGURE 123 Reference Listing (Part 2 of 2)

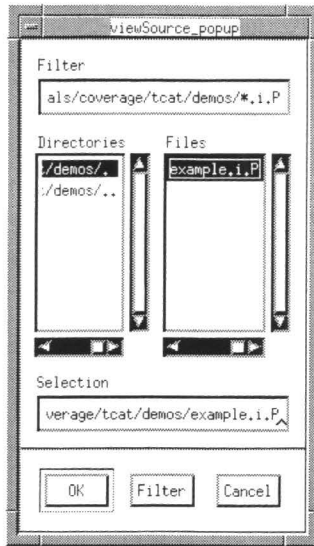


FIGURE 124 Source Viewing Pop-Up Window

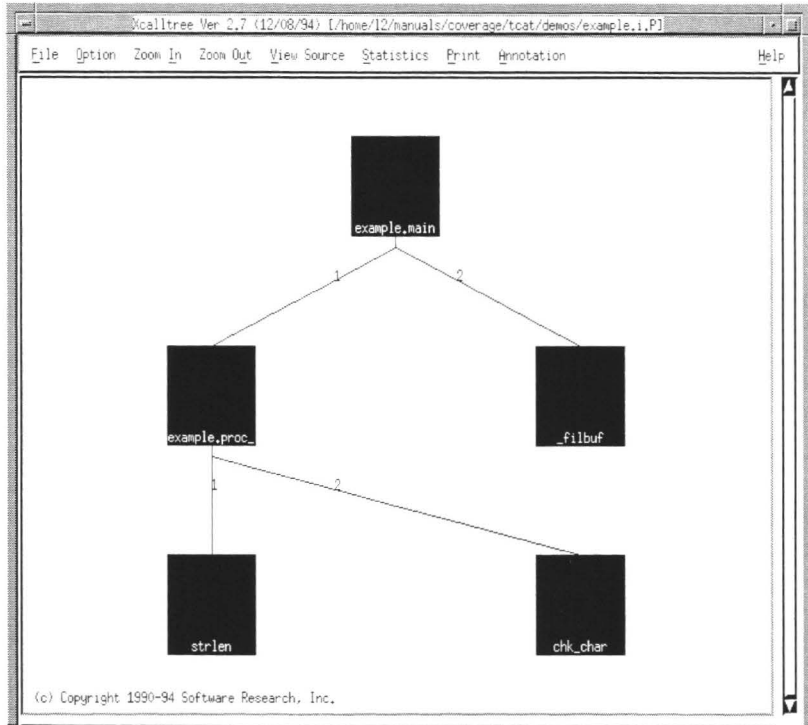


FIGURE 125 Source Viewing

Testing Guidelines: S-TCAT/C

This section presents some general guidelines that are of help during the rigorous software testing process imposed by *S-TCAT/C*.

The user should realize that these guidelines may have exceptions. The purpose of stating them is to establish some basic scale information.

1. The number of call-pairs in a candidate program is usually about 20 to 30 percent of the number of statements.
2. The number of tests required to achieve $S1 \geq 95\%$ (a minimum threshold for test completeness) tends to be about 25 to 50 percent of the number of call-pairs.
3. Typical programmers -- who do not have the benefit of detailed coverage analysis -- normally produce programs that are only 25 to 50 percent *S1* tested.
4. The execution-time overhead associated with instrumentation is in the neighborhood of 20 to 30 percent additional execution time and execution code. It can be higher if the program you are analyzing is very complex.
5. The trace files produced from your instrumented program should be moderate in length. If they become too large, then you should consider removing the instrumentation from some of the code.



System Restrictions and Dependencies

Certain restrictions exist in the way *S-TCAT/C* can be used. They are summarized here.

-
- It is important to recognize that *S-TCAT/C* can only be used with "legal" "C" programs. Non-legal constructions may pass through *S-TCAT/C*, but results cannot be guaranteed.
 - The function names `EntrMod`, `ExtMod`, `TCATFH`, `Strace`, and `Ftrace` are reserved for the runtime calls.
 - Both the instrumentor (`s-ic`) and system coverage analyzer (`scover`) take identifiers (function or variable names) that are up to 128 characters long.
 - Conditional expressions in "C" (of the form "*expr ? expr : expr*") are not supported; they must be expanded to the explicit "*if...[else]...*" form.



References

-
1. E. Kit, *State of the Art "C" Compiler Testing* , Tandem Computers, Inc., 1988.
 2. E. Uren, E. Miller, J. Irwin, *Automated Software Testing -- Case Studies* , IEEE Conference on Software Maintenance, Austin, Texas, September 1987.
 3. B. Boehm, *Software Engineering Economics* , Prentice-Hall, 1984.
 4. Software Research, Inc., *TCAT-PATH User's Manual* , 1989.
 5. W. G. Bently, E. F. Miller, *Ct Coverage -- An Initial Evaluation* , Conference Proceedings, Quality Week 1989, Software Research, Inc., San Francisco, California, May 1989.



Xdigraph Utility

Xdigraph is a utility which helps the user graphically understand a program's structure and flow.

22.1 Purpose

The **Xdigraph** utility draws digraphs, based on archive files from *TCAT* and *S-TCAT*. 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 1). Nodes are the places or "states" where the actions occur.

22.2 Xdigraph File Format

The format for a digraph chart file is very simple.

22.2.1 All digraph files:

- # in Col. 1 is a comment and is ignored (except # digraph...)
- Each line specifies an edge as a set of four strings: # digraph,tail, head, edge-name.
- The first blank line in any digraph definition region ends the scan for data (except if there are multiple-digraphs, explained below). Material after the first blank line is ignored.
- The topmost node in the display is always taken as the first-appearing node in the list of tail-nodes.
- If the digraph is ill-formed for any reason, then an error may result. **Xdigraph** tries to draw a picture in all cases.

22.2.2 Multiple digraph files:

- A multiple digraph file contains digraphs for many modules, each of which is identified with a formatted comment as follows:
- # digraph for 'module-name' in file 'file-name' This line precedes the digraph that corresponds to the module named "module-

name". There can be many digraphs in the file, each preceded by this line.

- When there are multiple digraphs in the submitted file, **Xdigraph** always draws a picture of the first-occurring one. You can select OTHER digraphs from the file using the **Load New Module** button from the **File** pull-down menu.

22.3 Invoking Xdigraph

To invoke **Xdigraph** from the command line, type:

```
xdigraph filename
```

Options:

```
[-A archive file]  
[-B filename]  
[-H filename]  
[-dig]  
[-h]
```

This will result in a digraph being drawn onscreen based on the filename given. The switches have the following values:

-A archive file	Archive file name. Default is 'Archive'.
-B filename	Spine file. This will change the text string for the digraph's nodes; the program will use the text settings for the filename typed after -B.
-H filename	Highlight specified file. File called will come up in "highlight" mode; program searches for file with <i>.pth</i> extension.
-dig	This switch specifies that the default <i>TCAT.dig</i> file will be drawn on the digraph.
-h	This switch brings up Xdigraph's help window.

You can also invoke the utility by simply typing its name:

```
xdigraph
```

In this instance, a blank main window will appear, and you can use the **File** pulldown menu to call up a file.

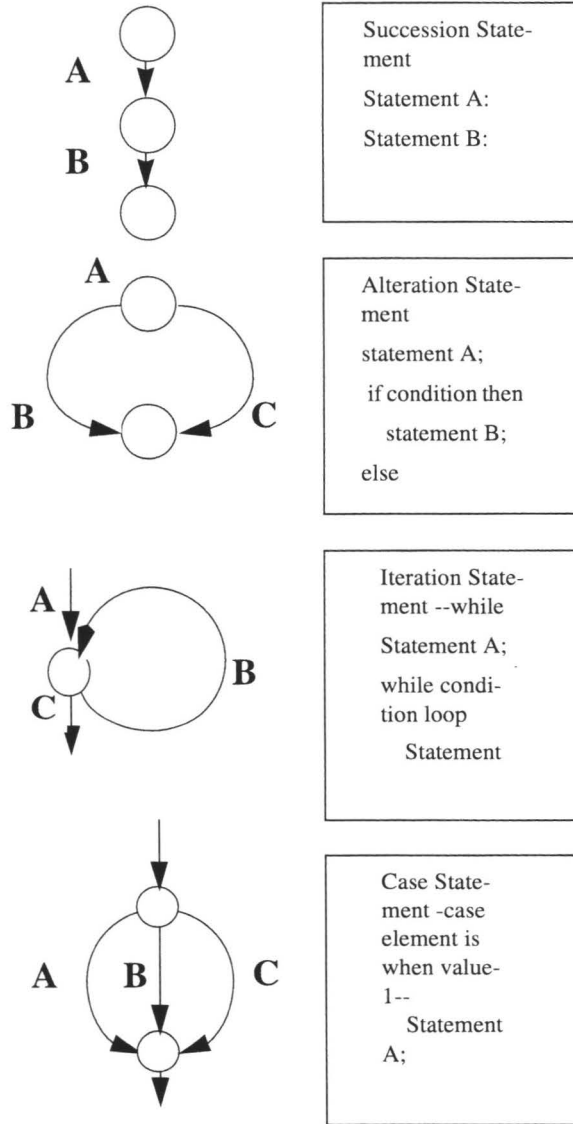


FIGURE 126 Program edges as represented in a digraph

22.4 Xdigraph Main Window

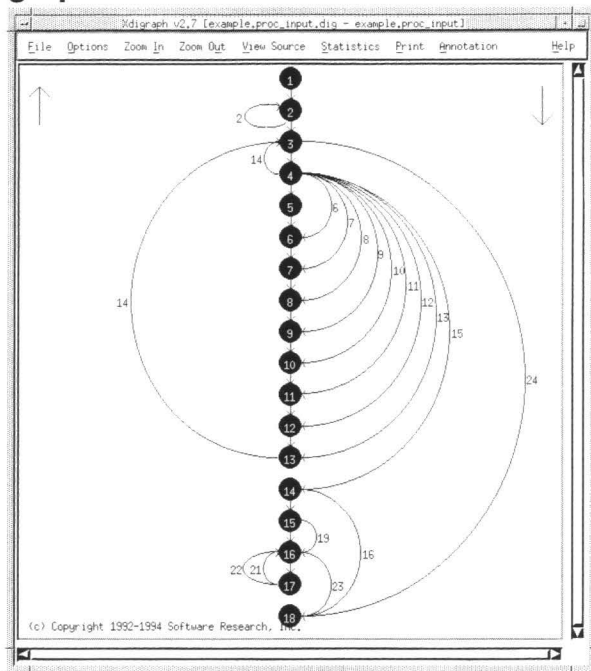


FIGURE 127 Xdigraph Main Window

Using **Xdigraph**, you can display a program's digraph and annotate it in a variety of ways. From **Xdigraph**'s Main Window menu bar, nine options are available.

22.4.1 File

This window allows you to select the file which will be displayed in the digraph.

22.4.2 Options

This window allows you to choose the characteristics of the nodes and edges displayed in the digraph, including shape, size, and color, as well as the scale for the **Zoom In** & **Zoom Out** options.

22.4.3 Zoom In

This window allows you to narrow the focus of the digraph, so that you can see it in more detail. There are maximum amounts that you can reduce or enlarge graphics, depending on what machine you are using.

22.4.4 **Zoom Out**

This window allows you to expand focus of the digraph, so that you can see it in wider perspective. There are maximum amounts that you can reduce or enlarge graphics, depending on what machine you are using.

22.4.5 **View Source**

This window allows you to view the source code for the current digraph.

22.4.6 **Statistics**

This window allows you to display pertinent statistics about the digraph, including node and edge counts, cyclomatic number, and path information.

22.4.7 **Print**

This window allows you to set the parameters and print the digraph.

22.4.8 **Annotation**

This window allow you to set the maximum and minimum thresholds for the nodes and edges in the digraph, as well as its path file.

22.4.9 **Help**

If you have a problem using **Xdigraph**, click on **Help**. Click your mouse on the **Action** pull-down menu and select **Search**. You will then get an **Enter String to search** dialog box. Click on the blank area and type the name of the option or function with which you need help.

NOTE: All these windows will be explained in further detail on the following pages.

22.5 File Pull-Down Menu

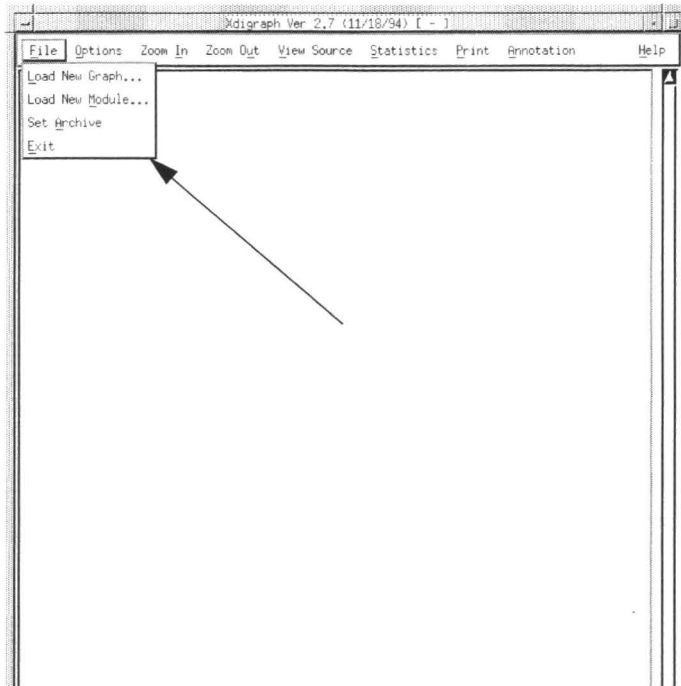


FIGURE 128 Digrph File Pull-Down Menu

22.5.1 Load New Graph

Click your mouse on the **File** pull-down menu shown above. Drag the mouse to **Load New Graph**. The File Message Box Pop-Up (Figure 129) will appear onscreen.

22.5.2 Load New Module

You use the **Load New Module** option if you have a multiple-digraph file and you want to choose a specific module in that file to be displayed.

When you click on this button the display shows you the set of available module names, taken from the multi-module digraph file that you have selected. You can then choose the module to be displayed. As soon as you click on **OK**, **Xdigraph** replaces the picture you have (if any) with the one corresponding to the named module.

If you don't have a multiple-module digraph file then this window may show no names. This is not an error but indicates that there are no module names specified.

22.5.3 **Set Archive**

The default Archive file is "Archive" but you can change this to any file you wish using the **Set Archive** option. After you push the button you will be given a file-selection popup. Select the file you want to use as the Archive file and click on **Apply** to confirm that choice. The current name of the Archive file is shown in the filename section of the window.

NOTE: The Archive file can have two formats, one for branch coverage (from *TCAT*) and one for call-pair coverage (from *S-TCAT*). It is important that the Archive file you are using reflects the kind of data appropriate for your display. Otherwise the annotation function will "fail" -- and the display will remain unannotated.

22.5.4 **Exit**

To close the current digraph window, select **Exit** from this pull-down menu.

22.5.5 Digraph File Message Box

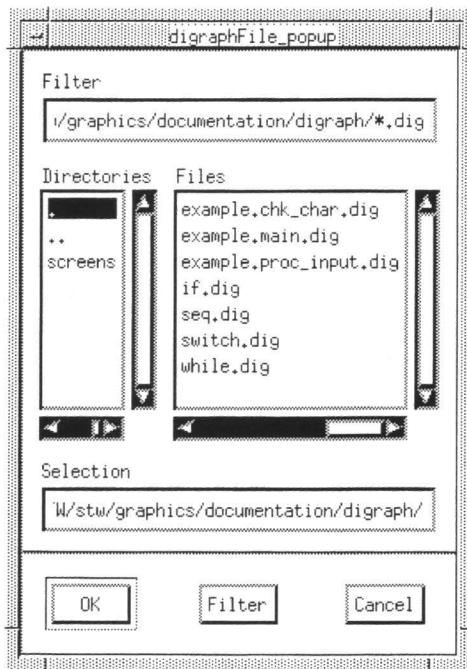


FIGURE 129 Digraph File Message Box

The message box in Figure above will pop up after you click the mouse on **Load New Graph** or **Load New Module**. The available options will allow you to choose the file to be represented in the digraph.

22.5.6 Filter

Allows you to limit the number of files that will be searched for; as above, only those ending in **.dig** will be included.

22.5.7 Directories

The directory from which the file is chosen to display in the digraph. Click on the chosen directory; it will show as darkened on the screen. Use the scroll bar at the bottom of this box if you cannot read the entire path-name of the directory.

22.5.8 Files

The actual file name selected to display in the digraph. Click on the file name, and the choice will be displayed in the **Selection** box. Use the scroll bar at the bottom of the box if you cannot read the entire filename.

22.5.9 Selection

Displays the file name selected in the **Files** box, or you can type in another name.

22.5.10 OK

Click **OK** when the desired file name is in the **Selection** box. The file named will then be represented as a digraph.

22.5.11 Filter Button

Clicking on this button will activate the filter limitations specified in the **Filter** box at the top of the window.

22.5.12 Cancel

To exit the window, without saving any changes, click on the **Cancel** button.

22.6 Options Window

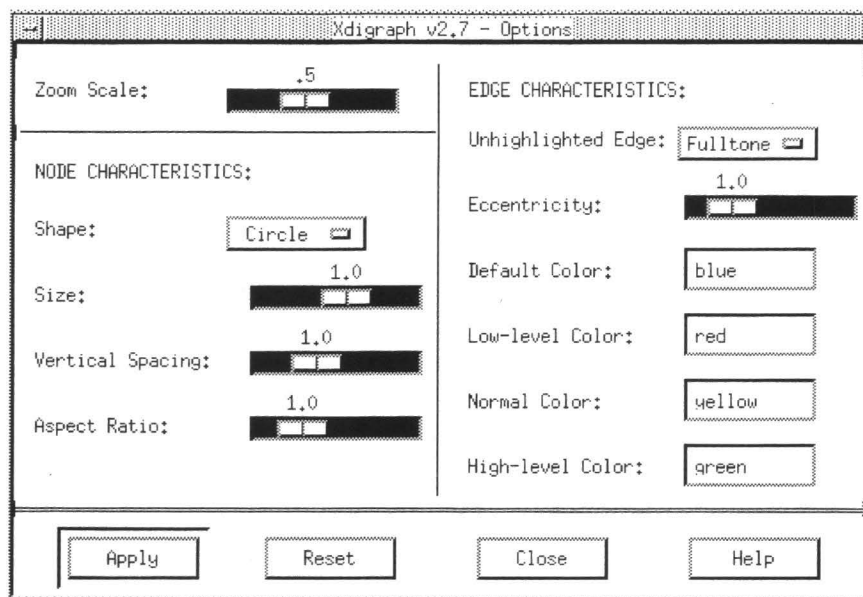


FIGURE 130 Xdigraph Options Window

This window will allow you to choose the scale for the **Zoom In** and **Zoom Out** options, the size of the digraph's nodes, and the colors of its edges.

22.6.1 Zoom Scale

This is the setting which affects the **Zoom In** and **Zoom Out** options. The default setting is 0.5, meaning a 50% reduction or enlargement is scale each time these buttons are used. To change the setting, move the slide rule left or right. Each 0.1 represents 10%, so if you slide the rule to .3, for example, the reduction and enlargements will be 30% each time. There are minimum and maximum amounts that you can reduce or enlarge graphics depending on what machine you are using.

22.6.2 Node Characteristics

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

- 22.6.2.1 Shape**
- You have four choices for shapes: **Circle**, **Box**, **Oval** or **Outlined** (the circle is drawn but not filled). The default setting is **Circle**.
- 22.6.2.2 Size**
- You can also choose the size of the circle, box or oval. The default size is 1.0.
- 22.6.2.3 Vertical Spacing**
- This is the amount of space between nodes. The default setting is 1.0.
- 22.6.2.4 Aspect ratio**
- The height-to-width ratio (for ovals or box shapes only). The default setting is 1.0.
- 22.6.3 Edge Characteristics**
- 22.6.3.1 Unhighlighted Edge**
- There are three choices: **fulltone**, **halftone** (dashes) or **blank** (no visible lines). The default setting is fulltone.
- 22.6.3.2 Eccentricity**
- Determines the curvature of the generated display. The default value is 1.0, meaning the edge between the two nodes is always drawn as a half-circle: bigger values make the picture wider, and smaller values narrower.
- 22.6.3.3 Default Color**
- Selects the basic color of the digraph's edges and nodes. The default setting is blue.
- 22.6.3.4 Low-level Color**
- In all cases, if the value of the chosen annotation is below the values indicated for Threshold 1, the display is done in the Low-level color. The default setting is red.
- 22.6.3.5 Normal Color**
- If the value of the chosen annotation is between Threshold 1 and Threshold 2, the Normal color is used. The default setting is yellow.

22.6.3.6 High-level Color

If the value of the chosen annotation is above the value stated in Threshold 2, then the High-level color is used. The default setting is green.

NOTE: If you have a monochrome display, then the three colors are expressed as a narrow, normal, and triple-wide line.

22.6.3.7 Apply

If you click on the **Apply** button, all the current settings in the **Options** window will be displayed on the digraph.

22.6.3.8 Reset

If you click on the **Reset** button, all the default settings will be restored to the **Options** window.

22.6.3.9 Close

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

22.6.3.10 Help

If you have a problem using the **Options** window, click on **Help**. Click your mouse on the **Action** pull-down menu and select **Search**. You will then get an **Enter String to search** dialog box. Click on the blank area and type the name of the option or function with which you need help.

22.7 Zoom In/Zoom Out Window

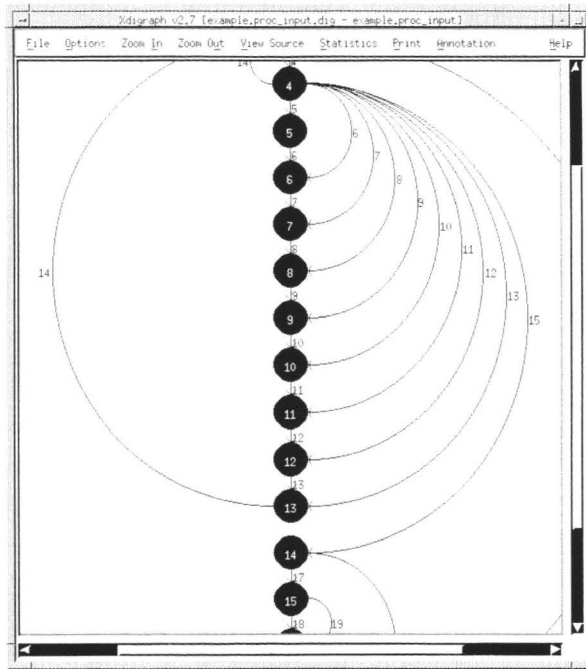


FIGURE 131 Zoom In feature illustrated

These buttons allow for a narrower or wider perspective of the digraph, depending on what you require. Click on the **Zoom In** button once to narrow the focus of the digraph, and click on the **Zoom Out** button to get a wider perspective of the digraph. Notice the difference between the digraph in Figure 6, after clicking on **Zoom In** once, and the same digraph, depicted in Figure .

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 viewing the digraph. For example, in Figure 6 above, to see the parts of the digraph below the node labeled 15, you would click your mouse on right-hand side scroll bar "down" arrow, and click on it as many times as necessary to get to the desired viewing point. You can also point, click and hold the mouse down to get to a certain area of the digraph.

NOTE: These features are limited by the display capabilities of your machine.

22.8 View Source Window

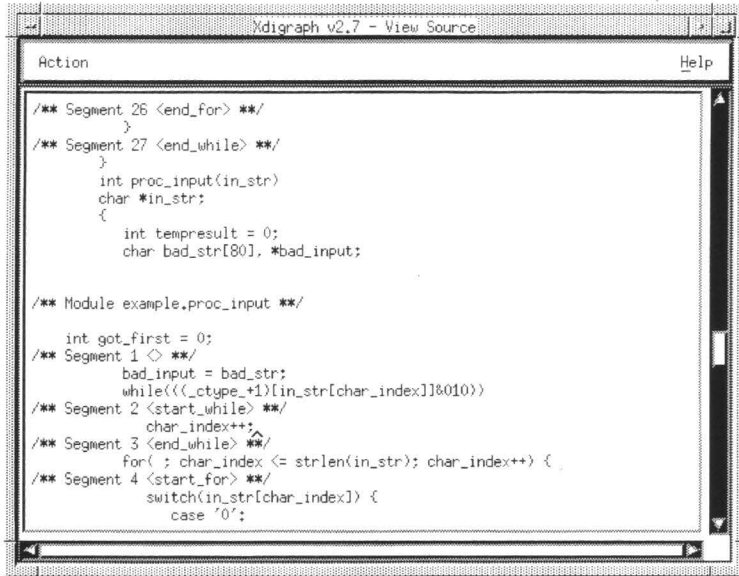


FIGURE 132 View Source Option Window

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, the **View Source** display will move to and highlight that particular edge's source code. The source code for the edge selected will appear in the middle of the window.

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.

22.9 Statistics Window

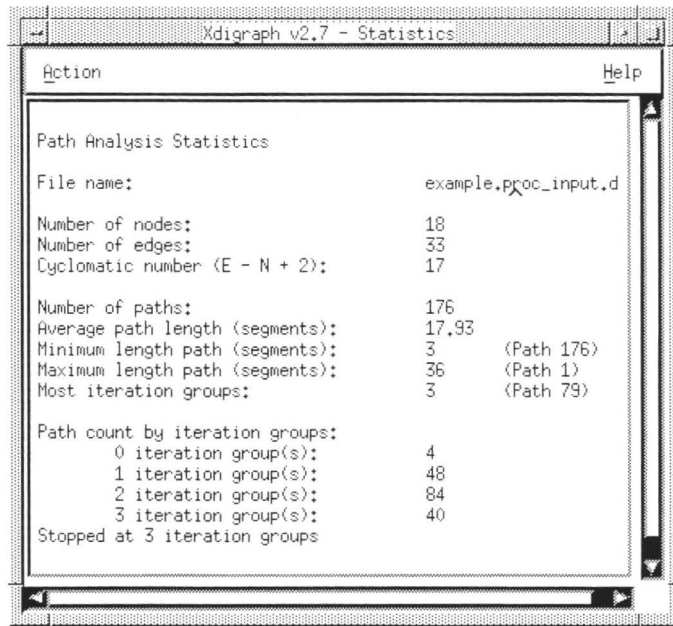


FIGURE 133 Statistics Option Window

This window displays the relevant statistical information for the digraph. If the file you are processing has multiple digraphs on it, then only the displayed digraph is reflected in the **Statistics** calculation.

WARNING: In some cases, particularly if the digraph is very complex, the **Statistics** calculation will take a long time. Practical internal limits have been set on the STW facility that computes these statistics (i.e. `apg` from `TCAT-PATH`) but even so the calculation may show the "hour glass" symbol.

When the limits are exceeded you will see the error message that results in the display where the statistics would ordinarily reside.

NOTE: The statistics generated in this window are always for the digraph that is on the display.

22.9.1 File Name

The name of the program studied in this particular digraph.

22.9.2 Node and Edge Count

The total number of nodes and edges in the digraph.

22.9.3 Cyclomatic Number (Cyclomatic Complexity)

A number which assesses program complexity according to the program's flow of control. This flow is based on the number and arrangement of decision statements in the code. The cyclomatic number can be calculated using the formula:

$$\text{cyclo} = e - n + 2$$

where **n** is the number of nodes in the graph, and **e** is the number of edges or lines connecting each node.

22.9.4 Average, Minimum and Maximum Path Lengths

The mathematical mean of all the paths in the program, as well as (user-defined) minimum and maximum possible lengths.

22.9.5 Path Count by Iteration Groups

The path count by iteration groups is the total number of distinct equivalence classes of program flow, figured using the one-trip loop assumption (for details on how this computation is done, see the *TCAT-PATH* manual).

The total path count has been shown to be very highly correlated with the overall effort required to completely test a module.

22.10 Print Window

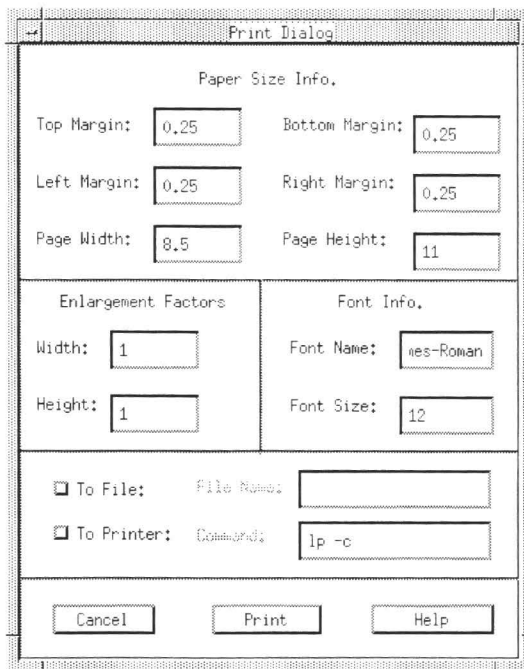


FIGURE 134 Print Dialog Window

The image you see will be printed to a standard print device. This window will allow you to configure it for your environment.

22.10.1 Paper Size Information

22.10.1.1 Top Margin

The distance from the top of the page to the first line. Default setting is 0.25 inches.

22.10.1.2 Left Margin

The distance from the left-hand side of the page to the first character of type. Default setting is 0.25 inches.

22.10.1.3 Page Width

The actual horizontal length of the paper you will be printing on. Default setting is 8.5 inches.

22.10.1.4 Bottom Margin

The distance from the bottom of the page to the last printed line. Default setting is 0.25 inches.

22.10.1.5 Right Margin

The distance from the right-hand side of the page to last character on the line. Default setting is 0.25 inches.

22.10.1.6 Page Height

Actual vertical measurement of the paper to be printed on. Default setting is 11 inches.

22.10.2 Enlargement Factors

22.10.2.1 Width/Height

The enlargement factors specify the size expansion, vertically or horizontally, to be applied to this particular print activity; in effect, the total number of 8.5 inch by 11.0 inch sheets onto which to draw the picture.

Selecting 1.0 means the picture will be printed on a single 8.5 inch x 11.0 inch sheet. Hence, width = 1.0 and height = 1.0 means to draw the image on a standard page.

If you change the Width to 2.0, for example, this means the picture will be drawn on two pages, i.e. in such a way that two 8.5 inch by 11.0 inch sheets can be pasted together to make a 17.0 inch by 11.0 inch image.

When more than one sheet is involved, the software numbers each page so that assembly into a larger diagram is simple and straightforward.

The software automatically sizes the image to fit into the smallest whole number of page equivalents. Also, the software sizes the diagram and the typefaces to "best fit" the specified size.

Some experimentation may be required to determine the optimum size for the diagram you are working with.

NOTE: The picture drawn on the printer always includes all of the information in the diagram, even if the entire diagram is not visible because of a zoom setting.

22.10.3 Font Information

22.10.3.1 Font name/Font size

The default font size, 12 pt, and the default font name, Times-Roman, normally provide good quality pictures. Times-Roman at 12 pt is commonly available on most printers.

You can choose different typesizes and type fonts depending on the sizes and fonts available on your computer.

22.10.4 Print locator

22.10.4.1 To File

Will create a PostScript (.ps) file, which you can use to have the digraph printed on any PostScript-compatible printer.

22.10.4.2 To Printer

You must name the printer to which the printing of the document will be sent.

When the printing has been sent to either a .ps file or a printer, a message box, **Print action completed**, will pop up. Click **OK** to close it.

NOTE: The print option requires use of a PostScript-compatible printer. If your machine is not attached to a PostScript compatible printer then the Print window option will be inoperative.

22.11 Annotation Window

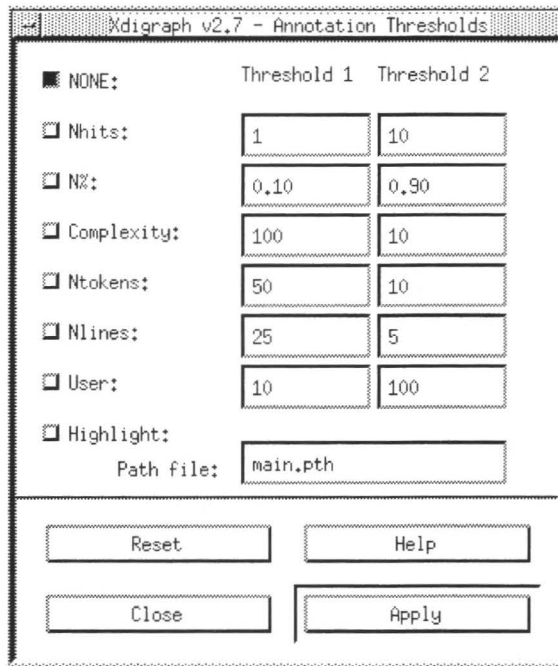


FIGURE 135 Annotation Thresholds Window

In many cases, annotation of the display is accomplished by showing the results of coverage testing, as reflected in the repository of multi-test coverage stored in the Archive file.

There are a number of ways to annotate the digraph. Typically this involves choosing a different color depending on where a particular parameter falls into user-specified ranges (thresholds).

There are five built-in annotation options and one user-defined annotation.

22.11.1 Threshold 1 & 2

Threshold 1 represents the lower limit, and Threshold 2 the upper limit desired for each annotation. The user can change the values of any threshold used by clicking in the window and typing in the new value. The values *won't* be applied to the current calltree unless you click the **Apply** button.

-
- 22.11.2 None**
No annotation; the digraph is left alone.
- 22.11.3 Nhits**
Number of times an edge is executed. The edge's color is based on this number. Default values: 1 10.
- 22.11.4 N%**
The relative number of times an edge has been executed. The color depends on this number's relation to the highest number of times any edge is exercised. Default values: 0.1 0.9.
- 22.11.5 Complexity**
Edge complexity value; the current value of the *METRIC*-produced edge complexity for this particular edge. Default values: 10 100.
-
- NOTE:** This setting is available only for *TCAT Ver 9* or later.
-
- 22.11.6 Ntokens**
Number of textual tokens on the edge. This number is a rough indicator of complexity, because it relates to the segment length (how many statements and how complex they are). Default values 10 50.
-
- NOTE:** This setting is available only for *TCAT Ver 9* or later.
-
- 22.11.7 Nlines**
The number of code lines associated with the edge. Default values: 5 25.
-
- NOTE:** This setting is available only for *TCAT Ver 9* or later.
-
- 22.11.8 User**
The outcome of calling a user-defined function, "*Xdigraph.user*", if that function exists along the current search path, is the value used to color the display. Default values: 10 100.
-

22.11.9 Highlight

The path highlight options permits you to see how a path set--typically one produced by **apg** (all paths generator)--applies to a particular digraph.

Each path in the set is shown highlighted. The path number is always shown "on screen". You can move forward or backward in the path set using the mouse buttons as follows:

- Left button: move down one path in the path set (N-1)
- Middle button: quit the highlighting activity.
- Right button: Move up one path in the path set (N+1)

22.11.10 Path File

This indicates the file you've selected to represent in the digraph.

22.11.11 Apply

If you click on the **Apply** button, all the settings changes made in the **Annotation Thresholds** window will be displayed on the digraph.

22.11.12 Reset

If you click on the **Reset** button, all the default settings will be restored to the **Annotation Thresholds** window.

22.11.13 Close

If you click on the **Close** button, you will exit the **Annotation Thresholds** window.

22.11.14 Help

If you have a problem using the **Annotation Thresholds** window, click on **Help**. Click your mouse on the **Action** pull-down menu and select **Search**. You will then get an **Enter String to search** dialog box. Click on the blank area and type the name of the option or function with which you are experiencing difficulty.

22.11.15 Colors

The colors of the digraph display are based on the annotation thresholds. They are selected in the Options window (see Section n.3 for further detail), and are used to distinguish the annotation to "low", "normal", and

"high". How these colors convey information is a function of which annotation is chosen.

NOTE: Whatever annotation option is selected for the digraph will be displayed in the upper left-hand corner of the main window, above an up-pointing arrow. In the example in Figure 11, the annotation is for **User**. If the **Zoom In** option is chosen, however, the corner where the annotation message is displayed may not be visible, and if **None** is the annotation chosen, no message will appear.

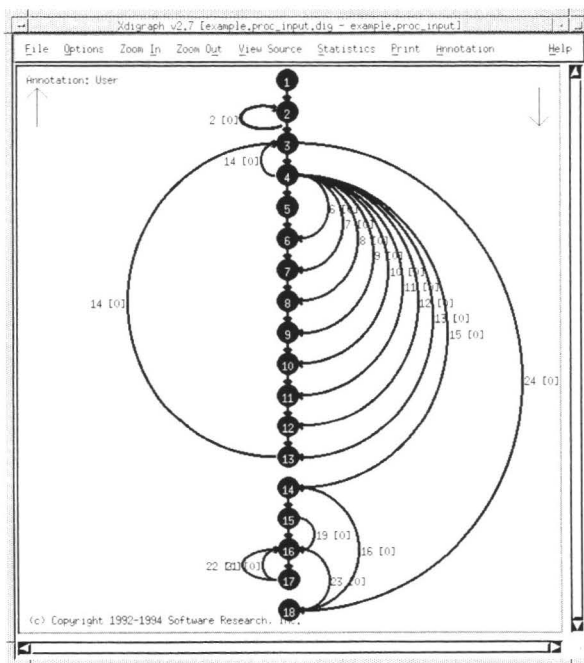


FIGURE 136 Sample Annotation for **User** Threshold

22.12 Quick Reference Guide to Xdigraph Annotations

Function	Display Coloring Reflects What Information?	Preset Low/High
Nhits	Absolute number of hits per edge (segment), from local Archive file. The Archive file is must be <i>from TCAT</i> . If not, silence.	1.0/ 10.00
N%	Percent of total hits in module for this edge (segment), from local Archive file. The Archive file must be from <i>TCAT</i> . If not, silence.	10.00/90.00
Complexity	Requires use of <i>TCAT</i> Ver 9.	100.00/10.00
Ntokens	Requires use of <i>TCAT</i> Ver 9.	50.00/10.00
Nlines	Requires use of <i>TCAT</i> Ver 9.	25.00/5.00
User	"(-1,0,1) = Xdigraph.user <i>N Lo Hi</i> " for all <i>N</i> = edge-number The default supplied sample script does something naive.	10.00 / 100.00
Highlight	Highlights <i>N</i> th path, beginning at <i>N</i> =1.(Path File) Left button moves up one path; right button moves down one path. Assumes <name>.pth file exists; apg generates the path list.	(Path File)

TABLE 1

Quick Reference Guide for Xdigraph Annotations

Xcalltree Utility

The **Xcalltree** utility displays the caller-callee dependence structure in a software program. The call tree is shown for the specified call-pair file--the one used when you invoke **Xcalltree**--and based on files created using the *TCAT* or *S-TCAT* tools.

23.1 Calltree Defined

A call-pair file's relationships are annotated on the calltree, and there are a number of ways to do this--ten built-in annotation options and one user-defined annotation. This information can then be displayed and printed out in a variety of ways.

23.2 Xcalltree File Format

The format for an **Xcalltree** chart file is very simple.

- # in Column 1 indicates a comment. There is no limit on the number of # comments in a file.
- The first blank line "ends the data". This means that the information describing a calltree chart must appear before the first blank line -- and that you can have no blank lines anywhere in the data region.
- After the first blank line, the rest of the file is treated as a comment.

A comment will be ignored by **Xcalltree**. A data line consists of a call-pair. The fields are separated by white spaces.

23.3 Invoking Xcalltree

Xcalltree can be invoked from the command line by typing:

```
Xcalltree filename  
[-D]  
[-r]  
[-m]  
[-h]
```

If you do this, the filename typed will be represented in the Xcalltree main window (see Figure 137). The switches have the following values:

-D	Maximum depth of calltree.
-r	Rootname for top-most file of calltree.
-m	Multigraph mode.
-h	This switch brings up the Xcalltree help window.

You can also simply type:

```
Xcalltree
```

A blank Xcalltree main window will appear. You would then select a file name from the **File** pull-down menu.

23.4 Xcalltree Main Window

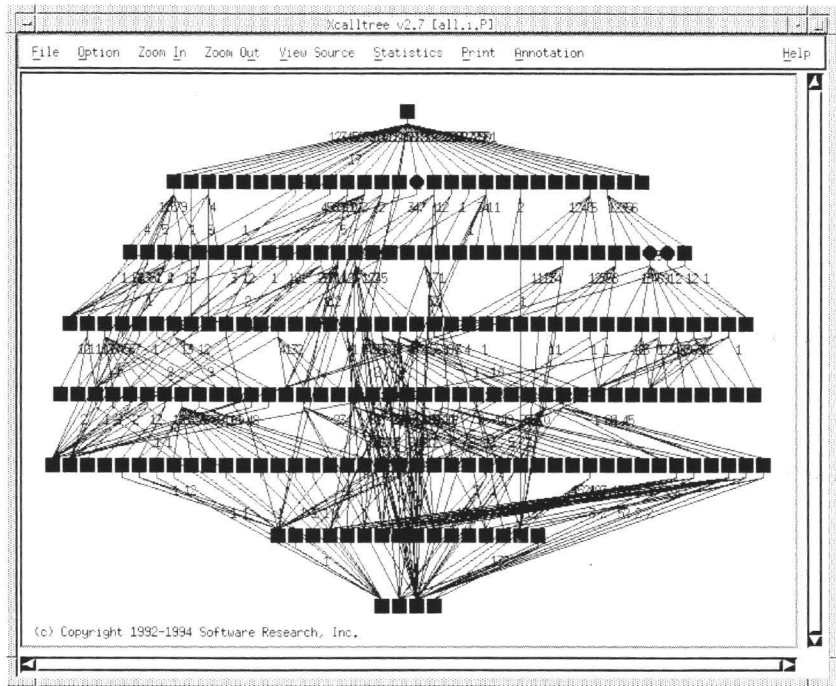


FIGURE 137 Xcalltree Main Window

Using **Xcalltree**, you can display a program's calltree and annotate it in a variety of ways. From **Xcalltree**'s main window menu bar, nine options are available.

23.4.1 File

This pull-down menu allows you to select the file which will be displayed in the calltree.

23.4.2 Options

This window allows you to choose the characteristics of the nodes and edges displayed in the calltree, including shape, size, and color, as well as the scale for the **Zoom In & Zoom Out** options.

23.4.3 Zoom In

This option allows you to narrow the focus of the calltree, so that you can see it in more detail. The amount you can **Zoom In** is limited to your display's capabilities.

23.4.4 Zoom Out

This option allows you to widen the focus of the calltree. The amount you can **Zoom Out** is limited to your display's capabilities.

23.4.5 View Source

This window allows you to view the source code for the current calltree.

23.4.6 Statistics

This window allows you to display pertinent statistics about the calltree, including links, number of callpairs, calltree depth, and number of recursive modules.

23.4.7 Print

This window allows you to set the parameters and print the calltree.

23.4.8 Annotation

This window allow you to set the maximum and minimum thresholds for the nodes and edges in the calltree, as well as its path file.

23.4.9 Help

If you have a problem using **Xcalltree**, click on **Help**. Click your mouse on the **Action** pull-down menu and select **Search**. You will then get an **Enter String to search** dialog box. Click on the blank area and type the name of the option or function with which you need help.

NOTE: All these windows will be explained in further detail on the following pages.

23.5 File Pull-Down Menu

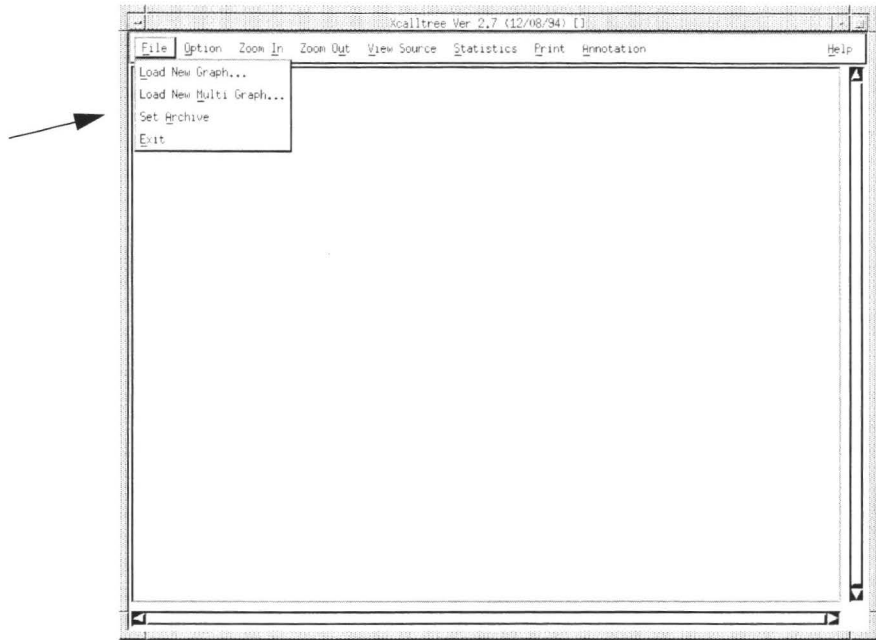


FIGURE 138 File Pull-Down Menu

23.5.1 Load New Graph

To display a calltree, click the mouse on the **File** pull-down menu. Drag the mouse to **Load New Graph** (Figure 138). The dialog box in Figure 139 will appear onscreen.

23.5.2 Load New Multi Graph

If there is more than one call between two nodes, the calltree will show *each* connection if **Load New Multi Graph** is selected. This may be difficult to see on a large calltree, so the example included in our demos directory is simple enough to see these connections.

23.5.3 Set Archive

The default Archive file is *Archive* but you can change this to any file you wish using the **Set Archive** button. After you push the button you will be given a file-selection popup. Select the file you want to use as the Archive file and click on **OK** to confirm that choice. The current name of the Archive file is shown in the filename section of the window.

NOTE: The Archive file can have two formats, one for branch coverage (from *TCAT*) and one for call-pair coverage (from *S-TCAT*). It is important that the Archive file you are using reflects the kind of data appropriate for your display. Otherwise the annotation function will “fail” -- and the display will remain unannotated.

23.5.4 Exit

If you wish to close the present calltree, drag the mouse to **Exit** on the pull-down menu, and the current calltree will disappear from the screen.

23.6 Calltree File Selection Dialog Box

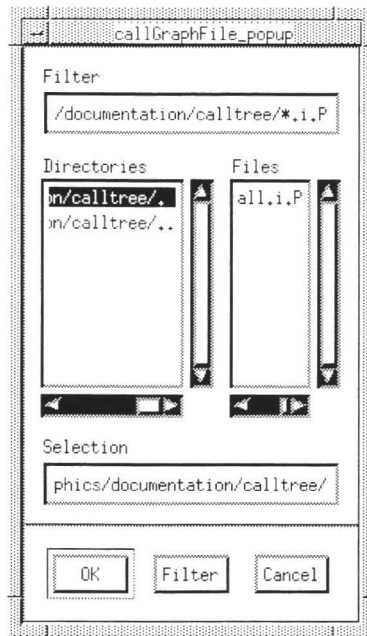


FIGURE 139 Calltree File Selection Dialog Box

This window pops up after you select **Load New Graph** or **Load New Multi Graph**, and allows you to select the file to be displayed as a calltree, using the following options:

23.6.1 Filter

Allows you to limit the number of files that will be searched for; as above, only those ending in **.i,P** will be included.

23.6.2 Directories

The directory from which the file is chosen to display in the calltree. Click on the chosen directory; it will be highlighted on the screen.

23.6.3 Files

The actual file name selected to display in the calltree. Click on the file-name, and the choice will be displayed in the **Selection** box.

23.6.4 Selection

Displays the file name selected in the **Files** box, or you can type in another name.

23.6.5 OK

Clicking on the **OK** button will cause whatever file is currently in the **Selection** box to be displayed in the calltree.

23.6.6 Filter

This button activates whatever filtering has been specified in the **Filter** box at the top of the window.

23.6.7 Cancel

To close the **File** dialog box without selecting a file for display, simply click on the **Cancel** button.

23.7 Option Window

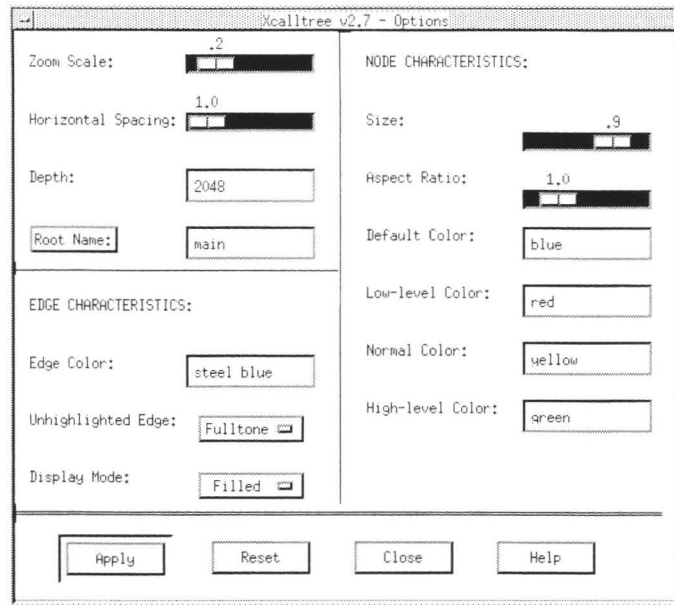


FIGURE 140 Option Window

23.7.1 Zoom Scale

The percentage for the **Zoom In** and **Zoom Out** functions. The default setting is 0.2, which means there will be a 20% enlargement or reduction. This value can be changed by sliding the ruler to the left (smaller) or right (larger). Each 0.1 is equal to 10%, thus setting the ruler to 0.4 would mean a 40% reduction or enlargement of the calltree each time you click **Zoom In** or **Zoom Out**.

23.7.2 Horizontal Spacing

The space between the nodes in the calltree. The default setting is 1.0.

23.7.3 Depth

The **Depth** value specifies the number of layers of the tree that will be displayed. The default value, 2048, is "very large" and it is unlikely that any

real-world calltree will be that deep. You would set the value to a smaller number, e.g. 10, if you want to limit the amount of detail on the screen. Using a smaller value for depth tells **Xcalltree** to disregard *all* calls below the specified value.

Also note that the **Connections** option can be adjusted to have a maximum upward and downward extent.

23.7.4 Root Name

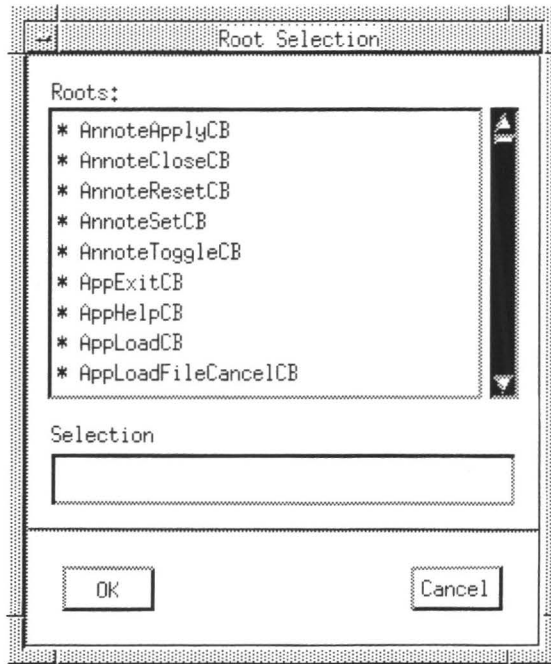


FIGURE 141 Root Name Selection Window Example 1

The calltree on the display is normally the first-occurring one in the callpairs file that **Xcalltree** processes. Some callpairs files contain more than one tree, i.e. more than one single “root” module name and the associated calls. If you want to view a *different* calltree than the one on the display, you do this by clicking on the **Root Name** button.

The resulting root-selection window is as shown in Figure 141. Every possible function name is shown in the list in the floating window.



FIGURE 142 Root Name Selection Window Example 2

Modules that are possible “roots” for the call tree, i.e. which are not called by another name in the file, are shown with a “~” (as in Figure 142). These are shown in alphabetical order at the top of the list.

Modules that *never* call another module are shown with a “~” in front of the name (as in Figure 144). They are sorted to the bottom of the list.

All other modules, those which are called by some root name or are in the downward chain from some root -- any one of which could be chosen as a new “root” name -- are in the middle of the list. Simply click on the name you wish to be the root and the new call-tree using that name is shown.

NOTE: If the Depth Value is set to a low number only PART of a tree may be visible.

23.7.5 Edge Characteristics

23.7.5.1 Edge Color

The actual color of the edge. Default setting is steel blue.

23.7.5.2 Unhighlighted Edge

The kind of unhighlighted edge to use: **Fulltone**, **Halfitone** (dashes), or **Blank** (no lines). Default setting is **Fulltone**.

23.7.5.3 Display Mode

Determines whether the nodes are darkened (**Filled**) or outlined (**Outline**). Default setting is **Filled**.

23.7.6 Node Characteristics

23.7.6.1 Size

The relative size of the box representing each nodule. Boxes are used for “normal” functions. Circles are used for self-referencing modules. Triangles are used for modules that are invoked recursively.

23.7.6.2 Aspect Ratio

The height-to-width ratio of the box.

23.7.6.3 Default Color

Selects the basic color of the calltree’s edges and nodes. The default setting is **blue**.

23.7.6.4 Low-level Color

In all cases, if the value of the chosen annotation is below the values indicated for Threshold 1, the display is done in the Low-level color. The default setting is **red**.

23.7.6.5 Normal Color

If the value of the chosen annotation is between Threshold 1 and Threshold 2, the Normal color is used (only when some edges are highlighted). The default setting is **yellow**.

23.7.6.6 High-level Color

If the value of the chosen annotation is above the value stated in Threshold 2, then the High-level color is used. The default setting is **green**.

NOTE: If you have a monochrome display, then the three colors are expressed as a narrow, normal, and triple-wide line.

23.9 View Source Window

```

Widget      editInputT1[12];
Widget      editInputT2[12];
Widget      thresholdLabel1;
Widget      thresholdLabel2;
Widget      annoteSeparator;
extern char  fileName[];
void UpdateAnnoteToggle(toggle, state)
int toggle, state;
{

/** Module UpdateAnnoteToggle **/

    int i;
    if (!state) {
        XmToggleButtonSetState(appAnnoteToggle[toggle], 1, 0);
    /** Call-pair 1 **/
    }
    else {
        XmToggleButtonSetState(appAnnoteToggle[currAnnoteToggle]
    /** Call-pair 2 **/
        currAnnoteToggle = toggle;
        GetColor(currAnnoteToggle, thresholds[currAnnoteToggle].t
    /** Call-pair 3 **/

```

FIGURE 144 View Source Window

23.9.1 Description of Source Code Viewing

The source-code text you see corresponds to the diagram. The text is positioned to show you the location of the call-pair you clicked on (or the first call-pair in the module, if you don't have a multi-graph on the screen). Also, if you click on the name of a function, **Xcalltree** will invoke **Xdigraph** and show you the detailed structure of that function. From **Xdigraph** you can view the source from that perspective, i.e. in terms of edges and nodes rather than call-pairs.

23.10 Statistics Window

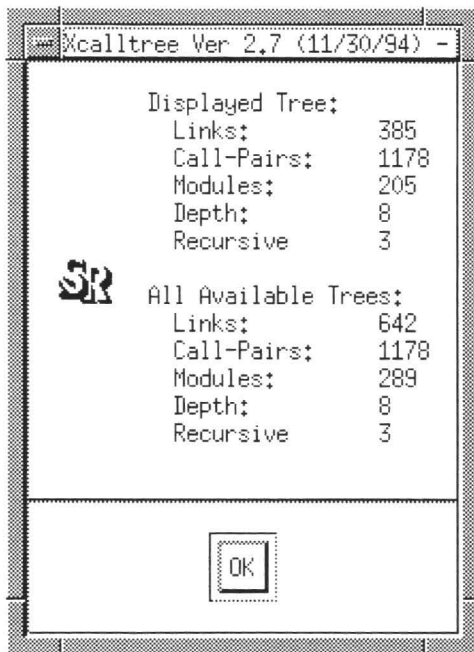


FIGURE 145 Statistics Window

The statistics you are given by **Xcalltree** are in two sections, the first pertaining to the calltree that you see on the screen, and the second pertaining to the entire file of information you supplied to the call to **Xcalltree**.

23.10.1 Links

This is the number of module-to-module connections in the diagram.

23.10.2 Call pairs

The total number of distinct, individual caller-to-callee connections in the diagram.

23.10.3 Modules/Depth

Modules is the total number of different names in the calltree, and depth indicates the maximum depth (either for links or for call-tree pairs).

23.10.4 Recursive

If the call tree is recursive, that is, if some module calls itself or calls some module for which there is a “self-referencing” chain, the number of such functions will be shown here.

23.11 Print Window

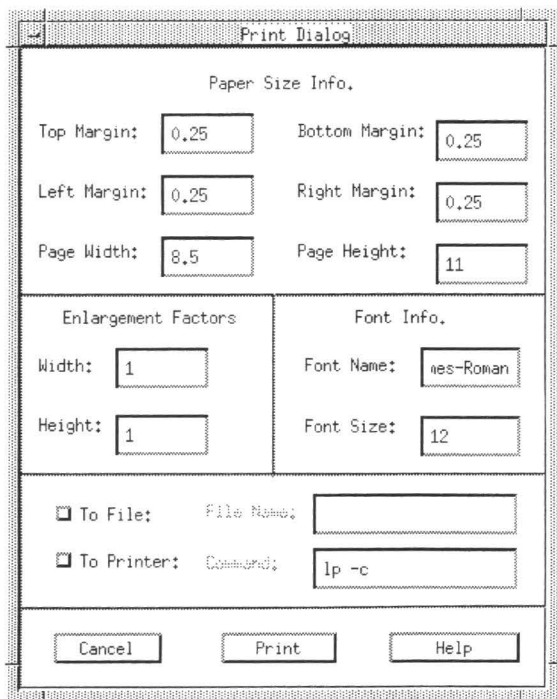


FIGURE 146 Print Window

The image you see will be printed to a standard print device. This window will allow you to configure the printing for your environment.

23.11.1 Paper Size Information

23.11.1.1 Top Margin

The distance from the top of the page to the first line. Default setting is 0.25 inches.

23.11.1.2 Left Margin

The distance from the left-hand side of the page to the first character of type. Default setting is 0.25 inches.

23.11.1.3 Page Width

The actual horizontal length of the paper you will be printing on. Default setting is 8.5 inches.

23.11.1.4 Bottom Margin

The distance from the bottom of the page to the last printed line. Default setting is 0.25 inches.

23.11.1.5 Right Margin

The distance from the right-hand side of the page to last character on the line. Default setting is 0.25 inches.

23.11.1.6 Page Height

Actual vertical measurement of the paper to be printed on. Default setting is 11 inches.

23.11.2 Enlargement Factors**23.11.2.1 Width/Height**

The enlargement factors specify the size expansion, vertically or horizontally, to be applied to this particular print activity; in effect, the total number of 8.5 inch by 11.0 inch sheets on which to draw the picture.

Selecting 1.0 means the picture will be kept on a single 8.5 inch x 11.0 inch sheet. Hence, width = 1.0 and height = 1.0 means to draw the image on a standard page.

If you change the width to 2.0, however, this means the picture will be drawn on two pages, i.e. in such a way that two 8.5 inch by 11.0 inch sheets can be pasted together to make a 17.0 inch by 11.0 inch image. When more than one sheet is involved, the software numbers each page (on the bottom center) so that assembly into a larger diagram is simple and straightforward. To assemble a diagram, start with sheet #1 in the lower left-hand corner.

The software automatically sizes the image to fit into the smallest whole number of page equivalents. Also, the software sizes the diagram and the typefaces to "best fit" the specified size.

Some experimentation may be required to determine the optimum size for the diagram you are working with.

NOTE: The picture drawn on the printer always includes all of the information in the diagram, even if the entire diagram is not visible because of a zoom setting.

23.11.3 Font Information

23.11.3.1 Font name/Font size

The default font size, 12 pt, and the default font name, Times-Roman, normally provide good quality pictures. Times-Roman at 12 pt is commonly available on most printers.

You can choose different typesizes and type fonts depending on the sizes and fonts available on your computer.

23.11.4 Print locator

23.11.4.1 To File

Will create a PostScript (.ps) file, which you can use to have the calltree printed on any PostScript-compatible printer.

23.11.4.2 To Printer

You must name the printer to which the printing of the document will be sent.

When a printing has been sent to either a .ps file or to a printer, a message window saying **Print action completed** will pop up. Click **OK** to close this window.

NOTE: The print option requires use of a PostScript-compatible printer. If your machine is not attached to a PostScript compatible printer then the Print window option will be inoperative.

23.12 Annotation Window

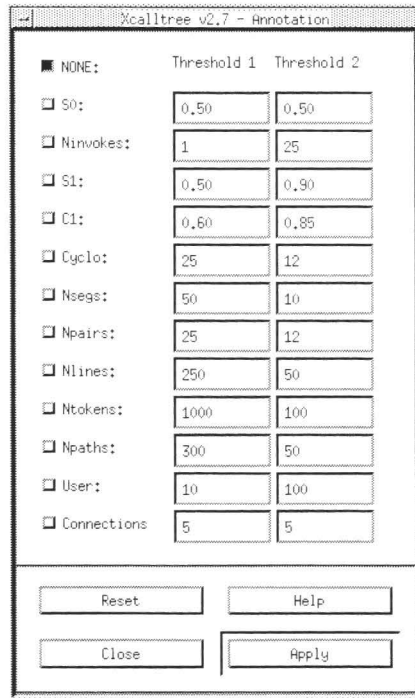


FIGURE 147 Annotation Window

Annotation of the display (using the **Annotations** button), in many cases is accomplished by showing the results of coverage testing, as reflected in the repository of multi-test coverage stored in the Archive file.

There are a number of ways to annotate the calltree. Typically this involves choosing a different color depending on where a particular parameter falls into user-specified ranges (thresholds).

There are ten built-in annotation options and one user-defined annotation.

23.12.1 Threshold 1 & Threshold 2

Threshold 1 represents the upper limit, and Threshold 2 the lower limit desired for each metric. You can change the values of any threshold used in the annotation of the call tree by clicking in the window and typing in the new value. The values WON'T be applied to the current calltree unless you click the **Apply** button.

23.12.2 None

No annotation is shown.

23.12.3 S0

The current value of the S0 metric is used to cover the display.

23.12.4 Ninvokes

The current number of invocations of the module is used to color the display.

23.12.5 S1

Call pair coverage. The current value of the S1 (module coverage) metric is used to color the display.

23.12.6 C1

Branch coverage. The current value of the C1 (module coverage) is used to color the display.

23.12.7 Cyclo

Cyclomatic complexity. The value of the cyclomatic complexity is used to color the display. For this annotation to work, you must choose a file with a **.dig** suffix.

23.12.8 Nsegs

The number of segments in the module is used to color the display.

NOTE: This annotation works only with *TCAT* Ver 9 or later.

23.12.9 Npairs

The number of call-pairs in the module is the metric used to color the display.

23.12.10 Nlines

Number of source lines. The number of non-blank lines in the module is the metric used to color the display.

NOTE: This annotation works only with *TCAT* Ver 9 or later.

23.12.11 Ntokens

The number of tokens (i.e. non-blank strings, or “words”) in the module is the metric used to color the display.

NOTE: This annotation works only with *TCAT* Ver 9 or later.

23.12.12 Npaths

The number of paths in the selected module, as computed by *apg*, is the value used to color the display.

23.12.13 User

User-defined function. The outcome of calling a user-defined function, “*Xcalltre.user*”, if it exists, is the value used to color the display.

23.12.14 Connections

The **Connections** option can be adjusted to have a maximum upward and downward extent.

23.12.15 Apply

After setting the desired thresholds, click **Apply** to display them in the current calltree.

23.12.16 Reset

To restore the default settings to the window, click on **Reset**.

23.12.17 Close

To exit the **Annotation** window, click on **Close**.

23.12.18 Help

If you have a problem using the **Annotation** window, click on **Help**. Click your mouse on the **Action** pull-down menu and select **Search**. You will then get an **Enter String to search** dialog box. Click on the blank area and type the name of the option or function with which you are experiencing difficulty.

NOTE: In certain cases, the annotation you select may not be displayed immediately, depending on the complexity of the calltree. In such instances the pointer will convert to a “clock” symbol, and you will have

to wait until it reverts to the pointer symbol for the annotations to take effect.

When annotating the calltree, you may attempt to annotate an object file that is supplied through *X* or the machine language, to which you will not typically have the source code. In the case where you click on a module of this type, the following message box will pop up:



FIGURE 148 Xcalltree “NOT DEFINED in reference file” message box

23.13 Quick Reference Guide to Xcalltree Annotations

Function	Display Coloring Reflects What Information?	Preset Low/ High
S0	Whether module was/wasn't invoked, from Archive file. Shows only two colors on display, low and high. The Archive file must be from <i>S-TCAT</i> . If not, silence.	50.00 / 50.00
Ninvokes	Number of times module was invoked, from Archive file. The Archive file is assumed to be one from <i>S-TCAT</i> . If not, silence.	1 / 25
S1	S1 value for module, from Archive file. Module name must appear in Archive ; else no default color. The Archive file is assumed to be one from <i>S-TCAT</i> . If not, silence.	50.00 / 90.00
C1	C1 value for module, from Archive file. Assumes module name appears in Archive ; else no color. The Archive file must be from <i>TCAT</i> . If not, silence.	60 / 85
Cyclo	Cyclomatic number retrieved from a call to app <name>.dig -X cyclo . This annotation requires that <i>TCAT-PATH</i> have been run and thus that the <name>.dig file for the module exists. NO error messages are given for <name>.dig's not found, but they keep the default color.	25 / 12
Nsegs	The number of segments in the module. Requires use of <i>TCAT</i> Ver 9.	50.00/10.00
Npairs	Number of call-pairs in module, from Archive file. The Archive file must be from <i>S-TCAT</i> . If not, silence.	50.00/ 10.00
Nlines	Number of source lines. The number of non-blank lines in the module is the metric used to color the display Requires use of <i>TCAT</i> Ver 9.	250.00 / 50.00
Ntokens	The number of tokens (i.e. non-blank strings, or "words") in the module is the metric used to color the display. Requires use of <i>TCAT</i> Ver 9.	1000 / 100

TABLE 2

Quick Reference Guide to Xcalltree Annotations

Function	Display Coloring Reflects What Information?	Preset Low/High
Npaths	Number of paths in module retrieved from apg <name>.dig -X npaths. This annotation requires that <i>TCAT-PATH</i> have been run and thus that the <name>.dig file for the module exists. NO error messages are given for <name>.dig's not found.	300.00 / 50.00
User	"(-1,0,1) = Xcalltree.user <i>N Lo Hi</i> " for all <i>N</i> = pair-number. The default supplied sample does something naive.	10.00 / 100.00
Connections	Up and Down +5, -5 callers/callees from clicked function.	

TABLE 2

Quick Reference Guide to Xcalltree Annotations

Index of Terms

Symbols

.dig file 104
.i.A file 104
.i.c file 104
.i.L file 104
.Xdefaults file 107
.dig file 24

A

-a option 169
a.out 109
ACTIONS menu 232
Analyze Window 38
Archive file 167, 217
archive file 116, 122, 215, 263, 265
Archive Files 220

B

bottom-up testing 7, 8
boundary conditions 193
branch coverage status 76
branch/segment coverage 2

C

C compiler 111
C1 and S1 instrumentation 162
C1 coverage 2, 76, 87, 158
C1 coverage value 75
C1 value 74, 77, 174, 176
call-pair coverage 2
Call-Pair Listing file 260
CAPBAK 11
-ce option 159

Change the report width to button 122
command line instructions 16
command line usage, TCAT 157
commands, MS-DOS 237
compile, modified program 73
compilers, C 111
compilers, UNIX 111
Compiling 28
compiling & running 3
compiling, instrumented program 108
Configuration File 234
configuration file processing 231
configuration file, S-TCAT 229
cost benefit analysis, use of S-TCAT 186
cover command 167
coverage analysis reports 261
coverage analysis tools 1
coverage analyzer 73
Coverage Analyzer Options 121
Coverage file, creation of 122
coverage reports 3
Coverage Reports, definitions 42
Cross Development 214
cross-compile 178
crun0 - Raw Tracefile 212
crun0.o 108
crun1 - Standard Tracefile 212
crun1.o 108
crun1.o file 30
cruna.o 109
cumulative coverage report 221
Cumulative Report 219
Cumulative report, defined 42
customizing TCAT 179
-cw 197

D

data structures 6
de- instrumentation feature 162
default runtimes 176, 239
default trace file name 110, 221
De-instrumented File Switch 169
De-instrumented Module List Switch 169
demos directory 14
development system 178
Directed Graph Listing 24
Directed Graph Listing file 104
directive processing 163
directives 162, 175
DOS, preprocessing rules 200
Dynamic 189
Dynamic Analysis 6, 189

E

embedded system 210, 211
embedded system usage 178
Error Listing 24
error rate prediction 11
example, instrumented program 245
example.c 270
example.c program 14, 20
example.i file 22
example.i.A 58
example.i.c 58
example.i.E 58
example.i.L 58
example.i.S 58
EXDIFF 11
Execute window 26
Execute window, invoking & using 106

F

-f file 169
file naming conventions, S-TCAT 209
file naming conventions, TCAT 161
filename.c file 260
FILES menu 232
-fl value 197
for statement 164
function call 196
function calls 3, 34, 53
function definition boundary 163

G

Generate list of functions not included in
report button 121
Generate list of functions with C1>
button 121
graphical user interface 89
GUI defaults 179
GUI Operation 89
GUI parameters 179
GUI) Tutorial 275
GUI, OSF /Motif style 89

H

-H 170
-H option 265
-h option 267
header information 58
-help 170
Help window 91
help, on-line 91
Hit and Not Hit reports 219
Hit Report 265
Hit report 75
Hit Report Switch 170
host 210

I

i.S file 104
ic instrumentor 158
if statement 164
Ignore Errors Switch 197
Instrument window 18
instrumentation 3, 24, 53, 58, 98
instrumentation, C1 & S1 162
instrumentation, completion of 104
instrumentation, single/multi modules 7
instrumented program 73
Instrumented source 161
Instrumented Statistics file 24, 68, 104
instrumenting 249
instrumenting, with make files 99
instrumentor 199, 241
Instrumentor command 101
Instrumentor options 101
instrumentor options 159

L

-l option 269
 Linear Histogram 267
 Linear Histogram Report Switch 169
 Linear Histogram report, listing 78
 linking , object files 109
 Linking the Application 32
 -lj 198
 Logarithmic Histogram report 269
 Logarithmic Histogram Report Switch 169
 Logarithmic Histogram report, listing 79
 logical branch 3, 98
 logical branch coverage 14
 logical branch marker 58
 logical condition 2
 logical path coverage 2

M

-m 170, 198
 -m6 198
 Make command 115
 make file 111
 make files 112, 175
 make utility 115
 Manual Analysis 6
 memory models, Microsoft C 212
 Microsoft C 200, 238
 Microsoft C 6.0 compiler 237
 Minimal Output Switch 170
 mkarchive Utility 70
 mkarchive utility 174
 mksarchive 225, 226
 module testing 2
 modulename.dig 58
 MS-DOS Runtimes 212
 MS-DOS runtimes 237
 MS-DOS, UNIX environments 202
 Multi-Tasking runtime 214

N

-n 198
 -N, -n 170
 New Archive File Name Switch 169
 New Archive name button 121
 Newly Hit Report 266
 Newly Hit report 76, 77

Newly Hit Report Switch 170
 Newly Missed Report 266
 Newly Missed report 77
 Newly Missed reports 219
 -NH 170
 -NH option 266
 -nl file 170
 -NM option 266
 Not Hit call-pair 254
 Not Hit report 76, 254
 Not Hit Report Switch 170
 Not Hit report, defined 42
 Not Hit reports 219
 null archive files 225

O

object code 73
 object modules 32
 Old Archive name button 121
 on-line help 91
 on-line help, S-TCAT 231
 OPTIONS menu 233
 OSF/Motif 89

P

-p option 263
 parsing, candidate source code 196
 parsing, source code 158
 passive "directives" 162
 Past Report 219, 263
 Past Test report 74
 percent coverage recommended 4
 Preprocessed source file 161
 Preprocessing 102
 preprocessing 22, 23, 57, 131, 249
 Preprocessing Source Code 57
 preprocessing source code 241
 preprocessing step 100
 Preprocessor command 101
 Preprocessor options 101
 Preprocessor output suffix 101
 program module 18
 program statistics 259

Q

Quick Start 13

quick.trc file 40
quiet 34
quiet runtime 34, 73, 108, 177, 212

R

rc file, S-TCAT 229
recommended amount of coverage 53
Reference Listing 24
Reference Listing File 120
Reference Listing file 254
reference listing file 3, 53
Reference Listing report 120
Reference Listing report, defined 42
Reference report, listing 80
reliability modeling 11
Resource files 179
runtime modules, description 177
runtime object module 30
runtime object modules 108, 110
run-time parameters, S-TCAT 234
runtime routines 211
runtime, forking 214
runtimes, MS-DOS 212
runtimes, non-standard 178

S

-h name 169
-l name 169
S1 call-pair coverage 195
S1 coverage 2, 192, 195, 219
S1 coverage, definition 191
S1 metric 261
S1 value 225, 263, 273
S1/C1 coverage, relationship 191
scover 217, 261
scover command 219
scover, syntax 220
Segment Count Listing file 24, 70, 104
Segment reference listing 161
Set File Name option 20, 130
Set Runtime Obj Module option 108
s-ic 196
Single- and Multiple-Module Testing 7
SMARTS 11
Sort report by module name button 122
special runtimes 178
special runtimes (UNIX) 213

Specify maximum file name length button,
defined 102
Specify maximum function name length but-
ton, defined 102
SR file 179
Static Analysis 6
S-TCAT ASCII menus 229
S-TCAT configuration file 234
S-TCAT configuration file, sample 236
S-TCAT example 245
S-TCAT.fns file 198
stcat.rc file 229
STW/Coverage 2
switch statment 164
syntax errors 57, 100, 131
System pull-down menu 16

T

-t 198
TCAT 2, 200
TCAT invocation window 16
TCAT, GUI-version 16
TCAT-PATH 2
test cases 4
testing methods 6
threshold, proper coverage 11
top-down testing 7, 8
Trace Descriptor 215
trace file 3, 24, 36, 110, 122, 157, 212, 215,
265
trace file name, default 110
trace file selection 40
trace file, defined 108
trace file, naming 118
trace files, complete listing 168
Trace.trc 110, 221
transferring trace files 210
T-SCOPE 2
Turbo C 200
tutorial 13

U

-u 198
UNIX compilers 111
UNIX Instrumentation 200
UNIX, preprocessing rules 200
UNIX/XENIX make file, example 113

unreachable code 211

V

variable type rules 6
View Report option 44
View Report window 123
View Source option 71
View Source window 48
viewing source code 48

W

-w 198
while statement 164
window manager 14

X

-x 198
X Window System 14
Xcalltree utility 192, 260
Xdefaults file 100
Xtcat 16

Z

-z 198
-Z file 172
-Z option 270

Segment Coverage Status:

All Segments Hit. C1 = 100%

4	5	6	10					
2	5	7	9	10	12	13	14	16
17	18	20	21	22	23	24	27	29
30	31	34	37	38	39	41	43	45
48	49	50	51	52	53	54	55	56
57	58	60	62	64	66	68	70	72
74	76	78	80	82	84	86	88	90
92	94	96	98	100	104	106	118	130
131	132	133	134	135	136	143	148	155
156								

1



Software Research

625 Third Street
San Francisco, CA 94107

STW/Coverage (Book 1)