(2)

IDA PAPER P-2769

AN EXAMINATION OF SELECTED SOFTWARE TESTING TOOLS: 1992

Christine Youngblut

Bill Brykczynski, *Task Leader*

December 1992

Prepared for
Strategic Defense Initiative Organization

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 92-042571

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Final

4. TITLE AND SUBTITLE
An Examination of Selected Software Testing Tools: 1992

5. FUNDING NUMBERS
MDA 903 89 C 0003

Task T-R2-597.21

6. AUTHOR(S)
Christine Youngblut

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2769

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
SDIO/SDA
Room 1E149, The Pentagon
Washington, D.C. 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution. January 13, 1993.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

This paper reports on the examination of 27 tools that provide for test management, problem reporting, and static and dynamic analysis of Ada code. It provides software development managers with information that will help them gain an understanding of the current capabilities of tools that are commercially available, the functionality of these tools, and how they can aid the development and support of Ada software. During the course of the examination, the static and dynamic analysis tools were applied to a sample Ada program in order to assess their functionality. The test management and problem reporting tools were also subject to a practical examination using vendor-supplied data. Each tool was then described in terms of its functionality, ease of use, and documentation and support. Problems encountered during the examination and other pertinent observations were also recorded. Available testing tools offer important opportunities for increasing software quality and reducing development and support costs. The wide variety of functionality provided by tools in the same category, however, and, in some cases, lack of tool maturity, require careful tool selection on behalf of a potential user.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Testing Tools; Static & Dynamic Analysis; Problem Reporting; Ada.	504
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

IDA PAPER P-2769

AN EXAMINATION OF SELECTED SOFTWARE TESTING TOOLS: 1992

Christine Youngblut

Bill Brykczynski, *Task Leader*

December 1992

IDA

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.21

PREFACE

This paper was prepared by the Institute for Defense Analyses (IDA) for the Strategic Defense Initiative Organization (SDIO) as a follow-on effort for Subtask Order T-R2-597.21, "Software Testing of Strategic Defense Systems." The objective of this subtask is to assist the SDIO in planning, executing, and monitoring software testing and evaluation research, development, and practice.

In support of this objective, IDA conducted an examination of 27 tools that support software testing. These tools provide for test management, problem reporting, and static and dynamic analysis of Ada code. This paper presents the results of the examination and provides software development managers with information on current capabilities of available testing tools.

This paper was reviewed by the following members of the IDA research staff: Dr. Robert J. Atwell, Dr. Dennis W. Fife, Dr. Randy L. Garrett, Ms. Deborah Heystek, Ms. Audrey A. Hook, Dr. Richard J. Ivanetich, Dr. Reginald N. Meeson, and Dr. Richard L. Wexelblat.

SUMMARY

Software testing is labor intensive and can consume over 50% of software development costs. Rarely is sufficient, effective testing performed as evidenced by the fact that a failure rate of 3 to 10 failures per thousand lines of code is typical for commercial software. Moreover, the cost of correcting a defect increases as software development progresses; for example, the cost of fixing a requirements fault during operation can be 60 to 100 times the cost of fixing that same fault during early development stages. Consequently, timely defect detection is important. Automated testing tools can alleviate these problems by providing managers with more insight into the progress of test activities, by reducing the traditionally manual nature of testing, and by encouraging the application of improved testing practices. Yet reviews of testing practices and tool usage reveal extremely poor exploitation of available testing tool support. Recent surveys of test practitioners indicate that there are few common test practices and only scattered tool usage.

Over 600 testing tools from some 400 suppliers were identified during the course of this study. From these, 27 tools were selected for examination. These tools support test management, problem reporting, and static and dynamic analysis of Ada code. Consideration of tools that are dependent on special hardware, limited to regression analysis, or form an integral part of a computer-aided software engineering (CASE) system was postponed for a later effort. Also, care was taken not to duplicate the tool assessment efforts of other groups. During the course of the examination, the static and dynamic analysis tools were applied to a series of Ada programs in order to assess their functionality. The test management and problem reporting tools were also subject to a practical examination using vendor-supplied data. Each tool was then described in terms of its functionality, ease of use, and documentation and support. Problems encountered during tool use and other pertinent observations were also recorded.

Significant findings from this study include the following:

- **Test management tools offer critically needed support for test planning and test progress monitoring.** This category of test tool is perhaps the latest to come to market. With the exception of reliability analysis tools, which are becoming more common, progress monitoring capabilities are infrequently available and primitive.

Nevertheless, the ability of these tools to manage a collection of test information is very valuable and the data available from the analysis of this information is urgently needed to support the documentation and management of test activities.

- **Problem reporting tools offer additional support for test management by providing insight into the status of software products and the progress of development activities.** These tools are primarily intended to support the tracking of identified software problems and the management of problem resolution. They also provide information on the status and quality of software products; in particular, they capture the data needed for software reliability modeling. This data can also provide valuable insights into the status and quality of the software development processes themselves, and so support continuous process improvement.
- **Available static analysis tools are essentially limited to facilitating program understanding and assessing characteristics of software quality. They provide some minimal support for guiding dynamic testing.** The types of defects traditionally found by static analysis tools are now routinely checked for by Ada compilers. Currently, complexity analysis, control flow analysis, and software browsing are the most common static analysis functions supported.
- **Although many needed dynamic analysis capabilities are not commonly available, tools are available that offer considerable support for dynamic testing to increase confidence in correct software operation.** Dynamic analysis is the principle method used for software validation and verification. Here automated support for the preparation of a test bed, generation of test data, and analysis of test results is urgently needed. Tools that provide this functionality will decrease the cost of testing by increasing the productivity of the human tester and increase software quality by supporting test data adequacy analysis and test repeatability. Tools that offer elements of this type of support are available.

Tools of similar types vary widely in the capabilities they provide and in characteristics such as tailorability and robustness. Existing testing tools are generally easy to use and supported by good documentation. There were instances during this study, however, where different tools gave different results when performing the same function, for example, calculating cyclomatic complexity. Moreover, some of the tools contained faults. While most failures were trivial, others rendered a tool unusable.

Available testing tools offer important opportunities for increasing software quality and reducing development and support costs. Even so, there are a number of specific problems with these tools and a lack of needed functionality that may handicap testing of Ada software:

- There is a lack of tool support for testing concurrent Ada software.
- There is a need for increased tool integration to provide more complete coverage of testing activities.

- There is a need for integration of testing tools into CASE systems to provide improved feedback into development activities.
- There is a lack of data on the cost effectiveness of particular test techniques and tools that can be used to encourage and guide tool use.

PART I STUDY OVERVIEW

1. INTRODUCTION 1-1
1.1 Purpose 1-1
1.2 Scope 1-1

2. STATE OF PRACTICE 2-1

3. TEST REQUIREMENTS AFFECTING TOOL USE 3-1
3.1 Affect of SDIO Software Test Requirements on Tool Use 3-1
3.2 Affect of the SEI Process Maturity Model on Tool Use 3-3
3.3 Affect of the Software Metrics Program on Tool Use 3-5

4. APPROACH AND METHODS 4-1
4.1 Tool Selection 4-1
4.2 Method of Examination 4-4
4.3 CASE System Support for Software Testing 4-5
4.4 Development Environment Support for Software Testing 4-5

5. TEST MANAGEMENT 5-1
5.1 Test Planning and Documentation 5-2
5.2 Requirements Mapping 5-3
5.3 Test Progress Monitoring 5-3
5.4 Productivity Analysis 5-5

6. PROBLEM REPORTING 6-1
6.1 Report Types and Details Captured 6-1
6.2 Import Capability 6-2
6.3 Reporting Capabilities 6-2
6.4 Standards Conformance 6-3
6.5 Distributed Architecture 6-3

7. STATIC ANALYSIS 7-1
7.1 Complexity Analysis 7-1
7.2 Data Flow Analysis 7-3
7.3 Control Flow Analysis 7-3
7.4 Information Flow Analysis 7-5
7.5 Standards Conformance Analysis 7-5
7.6 Quality Analysis 7-6
7.7 Cross-Reference Analysis 7-6
7.8 Browsing 7-7
7.9 Symbolic Evaluation 7-7
7.10 Specification Compliance Analysis 7-8
7.11 Pretty Printing 7-8

8. DYNAMIC ANALYSIS 8-1

8.1 Assertion Analysis 8-1
8.2 Coverage Analysis 8-2
8.2.1 Structural Coverage Analysis 8-2
8.2.2 Data Flow Coverage Analysis 8-3
8.2.3 Functional Coverage Analysis 8-4
8.3 Profiling 8-5
8.4 Timing Analysis 8-5
8.5 Test Bed Generation 8-5
8.6 Test Data Generation Support 8-6
8.6.1 Structural Test Data Generation 8-7
8.6.2 Functional Test Data Generation 8-7
8.6.3 Parameter Test Data Generation 8-8
8.6.4 Grammar-based Test Data Generation 8-8
8.7 Test Data Analysis 8-8
8.8 Dynamic Graph Generation 8-8

9. FINDINGS 9-1
9.1 Status of Available Tools 9-1
9.2 Significant Deficiencies 9-4

PART II TOOL EXAMINATION REPORTS

10. INTRODUCTION 10-1

11. AdaQuest 11-1
11.1 Tool Overview 11-1
11.2 Observations 11-3
11.3 Planned Additions 11-4
11.4 Sample Outputs 11-4

12. AutoFlow-Ada 12-1
12.1 Tool Overview 12-1
12.2 Observations 12-2
12.3 Planned Additions 12-2
12.4 Sample Outputs 12-3

13. DISTRIBUTED DEFECT TRACKING SYSTEM (DDTs) 13-1
13.1 Tool Overview 13-1
13.2 Observations 13-3
13.3 Recent Changes and Planned Additions 13-4
13.4 Sample Outputs 13-4

14. EXPERT DEBUGGING SOFTWARE ASSISTANT (EDSA) 14-1
14.1 Tool Overview 14-1
14.2 Observations 14-3

14.3 Sample Outputs 14-3

15. LDRA Testbed 15-1
15.1 Tool Overview 15-1
15.2 Observations 15-5
15.3 Planned Additions 15-7
15.4 Sample Outputs 15-7

16. Logiscope 16-1
16.1 Tool Overview 16-1
16.2 Observations 16-5
16.3 Planned Additions 16-6
16.4 Sample Outputs 16-6

17. MALPAS 17-1
17.1 Tool Overview 17-1
17.2 Observations 17-2
17.3 Planned Additions 17-4
17.4 Sample Outputs 17-4

18. QES/MANAGER 18-1
18.1 Tool Overview 18-1
18.2 Observations 18-3
18.3 Planned Additions 18-4
18.4 Sample Outputs 18-4

19. SoftTest 19-1
19.1 Tool Overview 19-1
19.2 Observations 19-3
19.3 Sample Outputs 19-4

20. SQA:Manager 20-1
20.1 Tool Overview 20-1
20.2 Observations 20-4
20.3 Recent Changes and Planned Additions 20-5
20.4 Sample Outputs 20-5

21. SRE TOOLKIT 21-1
21.1 Tool Overview 21-1
21.2 Observations 21-2
21.3 Sample Outputs 21-3

22. T 22-1
22.1 Tool Overview 22-1
22.2 Observations 22-4
22.3 Recent Changes and Planned Additions 22-5

22.4 Sample Outputs 22-5

23. T-PLAN 23-1
23.1 Tool Overview 23-1
23.2 Observations 23-4
23.3 Recent Changes and Planned Additions 23-5
23.4 Sample Figures 23-5

24. TBGEN and TCMON 24-1
24.1 Tool Overview 24-1
24.1.1 TBGEN Overview 24-1
24.1.2 TCMON Overview 24-2
24.2 Observations 24-3
24.3 Recent Changes 24-4
24.4 Sample Outputs 24-4

25. TCAT/Ada, TCAT-PATH, S-TCAT/Ada, TSCOPE, & TDGen 25-1
25.1 Tool Overview 25-1
25.1.1 TCAT/Ada and S-TCAT/Ada Overview 25-2
25.1.2 TCAT-PATH Overview 25-3
25.1.3 TSCOPE Overview 25-4
25.1.4 TDGen Overview 25-5
25.2 Observations 25-6
25.3 Recent Changes 25-7
25.4 Sample Outputs 25-7

26. TST 26-1
26.1 Tool Overview 26-1
26.2 Observations 26-3
26.3 Recent Changes 26-4
26.4 Sample Outputs 26-5

27. Test/Cycle and Metrics Manager 27-1
27.1 Tool Overview 27-1
27.1.1 Test/Cycle Overview 27-2
27.1.2 Metrics Manager Overview 27-4
27.2 Observations 27-6
27.3 Planned Additions 27-6
27.4 Sample Outputs 27-7

28. TestGen, QualGen, GrafBrowse, and the ADADL Processor 28-1
28.1 Tool Overview 28-1
28.1.1 ADADL Processor Overview 28-2
28.1.2 TestGen Overview 28-3
28.1.3 QualGen Overview 28-4
28.1.4 GrafBrowse Overview 28-4

28.2 Observations 28-5
28.3 Planned Additions 28-6
28.4 Sample Outputs 28-6

REFERENCES A-1

ACRONYMS AND ABBREVIATIONS B-1

GLOSSARY C-1

LIST OF FIGURES

Figure 2-1. Tool Usage Reported in Software Test Practices Survey 2-3
Figure 11-1. AdaQuest Unit Nesting Report 11-5
Figure 11-2. AdaQuest Branch Report 11-6
Figure 11-3. AdaQuest Coverage Test Run Report 11-7
Figure 11-4. AdaQuest Unit Coverage Report 11-8
Figure 11-5. AdaQuest Branch Coverage Detail Report 11-9
Figure 11-6. AdaQuest Branch Coverage Summary Report 11-10
Figure 11-7. AdaQuest Branch Coverage Report Showing Test Runs 11-11
Figure 11-8. AdaQuest Branch Coverage Not-Hit Report 11-12
Figure 11-9. AdaQuest Coverage History Detail Report 11-13
Figure 11-10. AdaQuest Coverage History Summary Report 11-14
Figure 11-11. AdaQuest Interval Test Run Report 11-15
Figure 11-12. AdaQuest Interval Timing Report 11-16
Figure 12-1. AutoFlow-Ada Page 1 of 6 Flowgraph for Function ALTERNATE 12-4
Figure 12-2. AutoFlow-Ada Page 2 of 6 Flowgraph for Function ALTERNATE 12-5
Figure 12-3. AutoFlow-Ada Page 3 of 6 Flowgraph for Function ALTERNATE 12-6
Figure 12-4. AutoFlow-Ada Page 4 of 6 Flowgraph for Function ALTERNATE 12-7
Figure 12-5. AutoFlow-Ada Page 5 of 6 Flowgraph for Function ALTERNATE 12-8
Figure 12-6. AutoFlow-Ada Page 6 of 6 Flowgraph for Function ALTERNATE 12-9
Figure 13-1. DDTs Sample Defect Report 13-5
Figure 13-2. DDTs Management Summary Report: Defect Reports 13-6
Figure 13-3. DDTs Management Summary Report: Defect Arrival and Repair Rate (All Levels) 13-9
Figure 13-4. DDTs Management Summary Report: Defect Arrival and Repair Rate (Sev. 1 & 2) 13-10
Figure 13-5. DDTs Management Summary Report: Sample Histograms 13-11
Figure 13-6. DDTs Management Summary Report: Bug Summaries 13-14
Figure 13-7. DDTs Management Summary Report: General Statistics 13-15
Figure 13-8. DDTs Examples of GUI Outputs 13-17
Figure 14-1. EDSA Threads View of Compilation Unit LL_TOKENS 14-4
Figure 14-2. EDSA Breaks View of Compilation Unit LL_TOKENS 14-5
Figure 14-3. EDSA Screen of Statement Traversal Using Data Flow of Variable I 14-6
Figure 14-4. EDSA Screen of Statement Traversal Using Control Flow in Unit LL_TOKENS 14-7
Figure 14-5. EDSA Annotations Example in Compilation Unit LL_TOKENS 14-8
Figure 14-6. EDSA Pebbling Example in Compilation Unit LL_TOKENS 14-9
Figure 15-1. LDRA Testbed Management Summary for LL_COMPILE 15-8
Figure 15-2. LDRA Testbed Static Call Tree of LL_COMPILE 15-14
Figure 15-3. LDRA Testbed Dynamic Call Tree of LL_COMPILE 15-15
Figure 15-4. LDRA Testbed Data Flow Analysis of LL_COMPILE 15-16
Figure 15-5. LDRA Testbed Information Flow Analysis for LLFIND 15-19
Figure 15-6. LDRA Testbed Complexity Analysis for LLFIND 15-20
Figure 15-7. LDRA Testbed System View McCabe's Complexity Measure 15-25

Figure 15-8. LDRA Testbed System View Knots Complexity Measure 15-26
Figure 15-9. LDRA Testbed Kiviat Graph for LLFIND 15-27
Figure 15-10. LDRA Testbed LCSAJ Analysis for LL_COMPILE 15-28
Figure 15-11. LDRA Testbed Cross Reference Analysis for LLFIND 15-30
Figure 15-12. LDRA Testbed Dynamic Analysis for LL_COMPILE 15-31
Figure 15-13. LDRA Testbed System View Statement Coverage 15-39
Figure 15-14. LDRA Testbed System View Branch Coverage 15-40
Figure 15-15. LDRA Testbed System View Test Path (LCSAJ) Coverage 15-41
Figure 15-16. LDRA Testbed Coverage Achieved Comparison 15-42
Figure 15-17. LDRA Testbed Active Flowgraph of READGRAM 15-43
Figure 15-18. LDRA Testbed Data Set Analysis for LLFIND 15-44
Figure 15-19. LDRA Testbed Profile Analysis 15-45
Figure 16-1. Logiscope Control Graph of Function LLFIND 16-7
Figure 16-2. Logiscope Textual Representation of Control Graph of Function LLFIND . 16-8
Figure 16-3. Logiscope Basic Counts for Function LLFIND 16-9
Figure 16-4. Logiscope Commented Listing for Function LLFIND 16-10
Figure 16-5. Logiscope Kiviat Graph of Function LLFIND 16-11
Figure 16-6. Logiscope Criteria Graph of Function LLFIND 16-12
Figure 16-7. Logiscope Kiviat Graph of All Components 16-13
Figure 16-8. Logiscope Overall Metrics Distribution for Program Length 16-14
Figure 16-9. Logiscope Overall Metrics Distribution for Cyclomatic Complexity 16-15
Figure 16-10. Logiscope Components per Metrics Category for Number of Statements 16-16
Figure 16-11. Logiscope Overall Criteria Distribution for Testability 16-17
Figure 16-12. Logiscope Overall Criteria Distribution for Simplicity 16-18
Figure 16-13. Logiscope Quality Report 16-19
Figure 16-14. Logiscope Excerpt from Default Quality Model 16-20
Figure 16-15. Logiscope IB Coverage of Function LLFIND 16-22
Figure 16-16. Logiscope DDP Coverage of Component BUILDRIGHT 16-23
Figure 16-17. Logiscope LCSAJ Coverage of Component BUILDRIGHT 16-25
Figure 16-18. Logiscope IB Coverage Histogram 16-28
Figure 16-19. Logiscope DDP Coverage Histogram 16-29
Figure 16-20. Logiscope Overall IB Coverage for Input test1.lex 16-30
Figure 16-21. Logiscope Overall DDP Coverage for Input test1.lex 16-31
Figure 16-22. Logiscope Metrics Table of Root 16-32
Figure 16-23. Logiscope Call Graph Path Testability of Root 16-32
Figure 16-24. Logiscope Call Graph Component Accessibility of Root 16-33
Figure 16-25. Logiscope Call Graph Calling/Called Components of Root 16-33
Figure 16-26. Logiscope Dynamic Call Graph of Root 16-34
Figure 16-27. Logiscope List of Call Graph Components per Level from Root 16-35
Figure 16-28. Logiscope PPP Coverage of Root 16-36
Figure 17-1. MALPAS Sample Pascal Code Illustrating MALPAS Analyses 17-5
Figure 17-2. MALPAS Intermediate Language Translation of Sample 17-6
Figure 17-3. MALPAS Control Flow Analysis of ADVANCE 17-8
Figure 17-4. MALPAS Data Use Analysis of ADVANCE 17-8
Figure 17-5. MALPAS Information Flow Analysis of ADVANCE 17-9

Figure 17-6. MALPAS Semantic Analysis of ADVANCE 17-10
Figure 18-1. QES/Manager Report Layout 18-5
Figure 18-2. QES/Manager Map of Master Driver 18-6
Figure 18-3. QES/Manager Problem Report 18-7
Figure 19-1. SoftTest Graph Entry Phase Input 19-5
Figure 19-2. SoftTest Variation Analysis Phase Output 19-9
Figure 19-3. SoftTest Test Synthesis Phase Output 19-12
Figure 19-4. SoftTest Functional Variation Coverage Matrix 19-17
Figure 19-5. SoftTest Test Case vs. Node Name Definition Matrix 19-19
Figure 19-6. SoftTest Cause-Effect Graph 19-21
Figure 19-7. SoftTest Functional Requirements Report 19-24
Figure 19-8. SoftTest 2167A Document Template 19-27
Figure 20-1. SQA:Manager Test Plan for ACTIII02PN 20-6
Figure 20-2. SQA:Manager Test Specification Report for Test Spec ACTIII02DS 20-7
Figure 20-3. SQA:Manager Test Case Report for Test Case INVPRN 20-8
Figure 20-4. SQA:Manager Test Procedure Report for Procedure CHKRUNS 20-9
Figure 20-5. SQA:Manager Software Items Report 20-10
Figure 20-6. SQA:Manager Test Tool Report 20-10
Figure 20-7. SQA:Manager Test Log Report 20-11
Figure 20-8. SQA:Manager Test Case Report for Test Case INVPRN 20-12
Figure 20-9. SQA:Manager Problems Table 20-12
Figure 20-10. SQA:Manager Fixed Problems Ready for ReTest 20-13
Figure 20-11. SQA:Manager Cost of Repair Table 20-13
Figure 20-12. SQA:Manager Cost of Testing 20-14
Figure 20-13. SQA:Manager Cost of Repair Graph 20-15
Figure 20-14. SQA:Manager Cost of Testing Histogram 20-15
Figure 20-15. SQA:Manager Reliability Analysis Table and Graph 20-16
Figure 20-16. SQA:Manager Failure Intensity Table and Graph 20-17
Figure 20-17. SQA:Manager Plot of Incidents by Symptom 20-18
Figure 20-18. SQA:Manager Plot of Problems by Severity 20-18
Figure 21-1. SRE Toolkit Generated Reliability Measures 21-4
Figure 21-2. SRE Toolkit Failure vs. Execution Time Plot 21-5
Figure 21-3. SRE Toolkit Initial Intensity vs. Execution Time Plot 21-6
Figure 21-4. SRE Toolkit Present Intensity vs. Calendar Time Plot 21-7
Figure 21-5. SRE Toolkit Completion Date vs. Failure Data 21-8
Figure 21-6. SRE Toolkit Testing Resource Usage Parameter Estimation 21-9
Figure 21-7. SRE Toolkit Reliability Demonstration Chart 21-10
Figure 21-8. SRE Toolkit Completion Date vs. Failure Intensity Output 21-11
Figure 21-9. SRE Toolkit Life Cycle Cost and Failure Intensity Objective Plot 21-12
Figure 22-1. T Sample SDF 22-6
Figure 22-2. T Software Description Verification 22-9
Figure 22-3. T Software Description Metrics 22-10
Figure 22-4. T Design Rule Verification 22-11
Figure 22-5. T Test Catalog 22-13
Figure 22-6. T Sample Generation 22-14

Figure 22-7. T Test Case Definitions 22-17
Figure 23-1. T-PLAN Test Model Functional Condition List Report 23-6
Figure 23-2. T-PLAN Test Model Sample Print for Input Ref 23-7
Figure 23-3. T-PLAN Test Model Input & Output References for Test Spec FIN 23-8
Figure 23-4. T-PLAN Test Model No Screen Data Testing for FIN 23-9
Figure 23-5. T-PLAN Test Model Output Print for FIN 23-9
Figure 23-6. T-PLAN Test Model Test Specification Information for FIN 23-10
Figure 23-7. T-PLAN Test Dictionary Function, Input, Output Reference Index 23-13
Figure 23-8. T-PLAN Test Dictionary Functions, Inputs, Outputs Used in FIN 23-14
Figure 23-9. T-PLAN Test Dictionary Condition Impact on Data Profiles 23-14
Figure 23-10. T-PLAN Test Dictionary Change Impact for Function MME, Input EIN, Output FIS 23-15
Figure 23-11. T-PLAN Test Dictionary Test Specification Index 23-15
Figure 23-12. T-PLAN Test Management Service Query Report for SQ 00002 23-16
Figure 23-13. T-PLAN Test Management Test Spec/SQ Log for FIN 23-16
Figure 23-14. T-PLAN Test Management Service Query Reports 23-17
Figure 23-15. T-PLAN Overall Progress for IBS 23-18
Figure 23-16. T-PLAN Test Management Service Query Reports 23-19
Figure 23-17. T-PLAN Test Management Reports 23-20
Figure 24-1. TBGEN Record File 24-5
Figure 24-2. TBGEN Trace File 24-6
Figure 24-3. TBGEN Generated Log File 24-7
Figure 24-4. TCMON Profile Execution Listing 24-8
Figure 24-5. TCMON Log File 24-9
Figure 24-6. TCMON Coverage Summary 24-10
Figure 25-1. TCAT/Ada Reference Listing for LL_COMPILE 25-8
Figure 25-2. TCAT/Ada Instrumentation Statistics for LL_COMPILE 25-9
Figure 25-3. TCAT/Ada Directed Graph for LLFIND from LL_COMPILE 25-10
Figure 25-4. TCAT/Ada Segment Coverage Report using test1.lex 25-11
Figure 25-5. TCAT/Ada Segment Coverage Report using test1.lex & sample.lex 25-14
Figure 25-6. TCAT-PATH Segment and Node Reference Listing for LL_COMPILE 25-18
Figure 25-7. TCAT-PATH Instrumentation Statistics for LLFIND 25-19
Figure 25-8. TCAT-PATH Cyclomatic Complexity of Function LLFIND 25-19
Figure 25-9. TCAT-PATH Segment Count for Each Module in LL_COMPILE 25-20
Figure 25-10. TCAT-PATH Digraph of Function LLFIND 25-20
Figure 25-11. TCAT-PATH All Paths for LLFIND 25-21
Figure 25-12. TCAT-PATH Basis Paths for LLFIND 25-21
Figure 25-13. TCAT-PATH Path Statistics for LLFIND 25-21
Figure 25-14. TCAT-PATH Path and Segment Information for LLFIND 25-22
Figure 25-15. TCAT-PATH Coverage Report for BUILDRIGHT using test1.lex 25-23
Figure 25-16. S-TCAT/Ada Call Graph for LL_TOKENS 25-24
Figure 25-17. S-TCAT/Ada Call-Pair Coverage using test1.lex 25-25
Figure 25-18. S-TCAT/Ada Call-Pair Coverage using test1.lex Accounting for All Call-Pairs 25-27
Figure 25-19. S-TCAT/Ada Call-Pair Coverage using test1.lex & sample.lex 25-29

Figure 25-20. TSCOPE Dynamic Display of Coverage on Directed Graph for LLFIND 25-33
Figure 25-21. TSCOPE Dynamic Display of Coverage Accumulation for LLFIND 25-33
Figure 25-22. TDGen Sample Value and Template Files 25-34
Figure 25-23. TDGen Table of Sequential Combinations for Initial Files 25-34
Figure 25-24. TDGen Output of First Random Execution 25-35
Figure 25-25. TDGen Output After 3 Executions with 1st Value File 25-35
Figure 25-26. TDGen Output After 2 Executions with 2nd Value File 25-35
Figure 26-1. TST Test Configuration File for Function LLFIND 26-6
Figure 26-2. TST Parameter Report for Function LLFIND 26-7
Figure 26-3. TST Execution History Report for Function LLFIND 26-9
Figure 26-4. TST Execution Summary Report for Function LLFIND 26-10
Figure 26-5. TST Sample Test Data File for Function LLFIND 26-11
Figure 26-6. TST Function LLFIND 26-12
Figure 27-1. Test/Cycle Requirements Hierarchy Report 27-8
Figure 27-2. Test/Cycle Requirement Description Report 27-10
Figure 27-3. Test/Cycle High-Level Validation Matrix Screen 27-11
Figure 27-4. Test/Cycle Intermediate Level Matrix Screen 27-11
Figure 27-5. Test/Cycle Detail Level Matrix Screen 27-11
Figure 27-6. Test/Cycle Build Description Report 27-12
Figure 27-7. Test/Cycle Components Description Report 27-13
Figure 27-8. Test/Cycle Test Run Description Report 27-14
Figure 27-9. Test/Cycle Requirements Validation Status Screen 27-15
Figure 27-10. Test/Cycle Test Run Validation Status Screen 27-15
Figure 27-11. Test/Cycle Test Case Description Report 27-16
Figure 27-12. Test/Cycle Test Case Linkages Screen 27-17
Figure 27-13. Test/Cycle Test Case Referenced by Requirement Screen 27-17
Figure 27-14. Test/Cycle Test File Description Report 27-18
Figure 27-15. Test/Cycle Work Request Description 27-19
Figure 27-16. Test/Cycle Work Request Log Report 27-19
Figure 27-17. Metrics Manager Database Full Report 27-20
Figure 27-18. Metrics Manager Enterprise & MIS Metric Summary Report 27-28
Figure 27-19. Metrics Manager Function Points Productivity vs. Type of Effort 27-30
Figure 27-20. Metrics Manager Development Defect Removal Efficiency vs. Size of Product 27-31
Figure 27-21. Metrics Manager Development Defect Removal Efficiency vs.Tools Used 27-32
Figure 27-22. Metrics Manager Development Unit Cost vs. Size of Product Showing Industry Data 27-33
Figure 28-1. ADADL Listing 28-7
Figure 28-2. ADADL Program Unit Cross Reference Report 28-8
Figure 28-3. ADADL Object Cross Reference Report 28-9
Figure 28-4. ADADL Type Cross Reference Report 28-10
Figure 28-5. ADADL Declaration Tree 28-11
Figure 28-6. ADADL Invocation Tree 28-12
Figure 28-7. ADADL Additional Cross Reference Reports 28-13

Figure 28-8. ADADL Complexity Summary Report 28-15
Figure 28-9. ADADL Program Unit ID Report 28-16
Figure 28-10. ADADL Objects Declared but Not Used Report 28-16
Figure 28-11. ADADL Types Declared But Not Used Report 28-16
Figure 28-12. ADADL Program Units Declared But Not Used Report 28-17
Figure 28-13. ADADL Program Units with High Complexity Metrics Report 28-18
Figure 28-14. ADADL Error Cross Reference Report 28-18
Figure 28-15. TestGen Test Conditions for Path Testing of LLFIND 28-19
Figure 28-16. TestGen Test Case Effort Report 28-21
Figure 28-17. TestGen Unreachable Statement Report for LL_COMPILE 28-22
Figure 28-18. TestGen McCabe Complexity Report for LL_COMPILE 28-23
Figure 28-19. TestGen Test Coverage Summary using test1.lex 28-24
Figure 28-20. TestGen Sub-Program Invocation Count Report using test1.lex 28-25
Figure 28-21. TestGen Statement Execution Report using test1.lex for ADVANCE 28-26
Figure 28-22. TestGen Branch Path Coverage Analysis using test1.lex for ADVANCE 28-27
Figure 28-23. TestGen Structured Testing Path Coverage Analysis using test1.lex for ADVANCE 28-28
Figure 28-24. TestGen Test Coverage Summary using test1.lex & sample.lex 28-30
Figure 28-25. QualGen Report Excerpt 28-31
Figure 28-26. GrafBrowse Flat Invocation Graph of LL_COMPILE 28-34
Figure 28-27. GrafBrowse Declaration Tree of LL_COMPILE 28-35
Figure 28-28. GrafBrowse Flat Callby Tree of LLFIND 28-36
Figure 28-29. Grafbrowse Browsing LLFIND 28-37

LIST OF TABLES

Table 2-1. Practices Reported in Software Test Practices Survey 2-1
Table 2-2. Practices Reported in Software Measures & Practices Benchmark Survey .. 2-2
Table 3-1. SDIO Test Requirements 3-2
Table 3-2. PMM-Implied Test Requirements 3-4
Table 3-3. Software Metrics Plan Implied Test Requirements 3-6
Table 4-1. Tools Examined in the CRWG and STSC Studies 4-2
Table 4-2. Tools Examined in the IDA Study 4-3
Table 4-3. Tools Planned for Future Examination 4-4
Table 4-4. CASE-based Testing Support 4-7
Table 4-5. Ada Development Environment-based Testing Support 4-8
Table 5-1. Test Management Capabilities of Examined Tools 5-1
Table 6-1. Problem Reporting Capabilities of Examined Tools 6-1
Table 7-1. Static Analysis Capabilities of Examined Tools 7-1
Table 7-2. Supported Complexity Measures 7-2
Table 8-1. Dynamic Analysis Capabilities of Examined Tools 8-1
Table 8-2. Structural Coverage Analysis Characteristics 8-4
Table 8-3. Test Bed Generation Characteristics 8-6
Table 10-1. Tool Profiles 10-2
Table 10-2. Supplier Profiles 10-4

PART I

STUDY OVERVIEW

1. INTRODUCTION

1.1 Purpose

This report provides software developers with information that will help them gain an understanding of the types of software testing tools that are available, the functionality of these tools, and how they can aid the development and support of Ada software for the Strategic Defense Initiative (SDI).

1.2 Scope

Tools are available to support a variety of testing tasks at different stages in the software life cycle. To make best use of available resources, the work described here was initially limited to the examination of tools that support the static and dynamic analysis needed for testing Ada code. Code-based testing was selected as being one area where automated support is critically needed, both to increase software reliability and to reduce development and support costs. Restriction to the Ada programming language [ANSI/MIL-STD-1983] was adopted in view of Department of Defense (DoD) Instruction 5000.2 [DoDI 1991]. The scope of the study was subsequently extended to include test management and problem reporting tools. The purpose of this extension is to accommodate DoD's increasing trend towards the use of software metrics to support the management of software development and as a basis for continual process improvement.

The report is divided into two parts. Part I starts by setting the scene for the following discussions by taking a brief look at the current state of practice in software testing. Special software test requirements imposed by the Strategic Defense Initiative Organization (SDIO), and how automated test tools could support meeting these requirements, are also discussed. Part I goes on to describe how particular tools were selected for examination, identifies the tools so selected, and outlines the method of examination. The following sections summarize tool functionality in the areas of test management, problem reporting, static analysis, and dynamic analysis. This first part of the report concludes by summarizing the findings resulting from this work.

Based on the experience gained during their examination, Part II provides a usage-based description of the tools and example report outputs. This more technical presentation is intended to provide further insight for the potential tool user.

This is a follow-on report to IDA Paper-2686 [Youngblut 1991]. The earlier report discusses the examination of some 10 tools for the static and dynamic analysis of Ada code. For convenience, those discussions have been updated as appropriate and are included here.

2. STATE OF PRACTICE

The high cost of software testing has long been recognized by the software community. In the early 1970s, data collected during development of a number of large software systems (e.g., SAGE, NTDS, Gemini, Saturn V, and IBM OS/360) revealed that 50% of development costs were incurred by software testing [Boehm 1980].[1] This figure holds true today [AFSCP 1987, Korel 1991, Yourdon 1990]. Even with this level of effort, operational software still fails. Commercial software typically experiences 3 to 10 failures per thousand line of code (KLOC) and industrial software experiences 1 to 3 failures per KLOC [Boehm 1988].

Recent surveys of current testing practices help to explain these figures. The Software Test Practices Survey [SQE 1990], conducted at the Seventh International Software Testing Conference, for example, found that software test practices were weak at the unit testing level and only slightly better for system and acceptance testing. In fact, when common testing practices were defined as those which more than 60% of respondents ranked as standard practice, *no* common practices for unit testing could be identified. Table 2-1 shows the percentage of responses indicating testing process and management practices as standard.

Table 2-1. Practices Reported in Software Test Practices Survey

Percentage of responses indicating practices common or standard	Unit Test	System Test	Acceptance Test	
Process Practices				
Software risks are systematically analyzed.	11	30	32	
Test cases & procedures are formally documented.	25	52	**60**	
Test are specified before software design.	6	15	15	
Test cases & procedures are saved after testing.	29	58	54	
Formal report of test results is produced.	33	**65**	**60**	
Requirements coverage is analyzed or traced.	17	55	54	
Code coverage is analyzed or traced.	18	28	27	
Design coverage is analyzed or traced.	15	32	38	
Formal exit criteria used to specify test completion.	22	18	17	
Tests are rerun after software changes.	18	39	38	
Test process is systematic and standardized.	39	**70**	**65**	
Test cases & procedures assigned unique names.	21	55	54	
Management Practices				Boldface print indicates common practice (>60%)
A record of time spent on testing is produced.	14	42	39	
Cost of testing is measured and reported.	11	28	24	
A record of faults and defects found is produced.	26	**68**	**70**	
The patterns of faults and defects regularly analyzed.	10	27	24	
Defect density is measured.	10	19	16	
User or customer satisfaction is measured.	17	42	42	
Number of changes or change requests is measured.	8	16	16	
Test effectiveness and efficiency measured & reported.	9	20	21	
Testers are formally trained.	10	30	23	
The test process is documented in standards manual.	24	22	30	

1. For NASA's Apollo program, 80% of the total software development effort was incurred by testing [Dunn 1984].

Xerox Corporation and Software Quality Engineering conducted a joint survey called the Software Measures and Practices Benchmark Study [SQE 1991]. The first part of this work provided a preliminary assessment of typical software practices and measures in use in industry. The results of this initial work were used to identify those organizations that employ the most of what industry generally considers to be good practices. The organizations selected were AT&T, E.I. DuPont de Nemours, GTE Corporation, IBM, NCR Corporation, Siemens AG, and Xerox Corporation. Each was asked to pick one or two of their "best" projects from which to provide data for the survey. Table 2-2 reproduces some of the results. Even though these organizations were selected as ones that most frequently employ advanced testing practices, very few testing practices were in common use at that time.

Although tools are more frequently used for system and acceptance testing rather than unit testing, the Software Test Practices Survey found that there were no types of tools that more than 60% of respondents cited as commonly used. As shown in Figure 2-1 the most widely used type of tool was only used by 50% of the respondents. Similarly, the Software Measures and Practices Benchmark Study found only scattered use of tools.

Table 2-2. Practices Reported in Software Measures & Practices Benchmark Survey

Mean scores for practical usage	Low	Mean	High
Process Practices			
Software risks (potential failures) are systematically analyzed.	1.00	1.57	1.94
Test planning & specifications are stated in requirements phase.	.90	1.47	2.14
Unit test plans/specifications are prepared.	1.19	1.96	**2.53**
Someone other than programmer performs/reviews unit testing.	.87	1.54	**2.59**
Module or program complexity is measured.	.46	1.16	1.81
Software changes are analyzed for ripple effect and test impact.	1.47	1.97	**2.33**
Unit branch & statement execution coverage is analyzed.	.60	1.24	1.49
Unit test results are recorded.	1.06	1.89	**2.71**
Tests are cross-referenced to requirements.	1.00	1.65	2.13
Test plans & specifications are formally reviewed.	1.91	2.21	**2.56**
Code coverage is analyzed for entire system during system test.	.68	1.51	2.14
Random testing is used to evaluate reliability.	1.00	1.72	**2.26**
Tests are systematically saved & reused.	1.10	2.18	**2.83**
Features fixed in previous test cycles systematically retested.	1.70	2.21	**2.67**
Management Practices			
Cost of quality activities is measured and reported.	.57	1.37	**2.41**
Defects are analyzed to determine cause & when created.	1.00	1.55	2.02
Defects found during testing are recorded & tracked.	**2.42**	**2.70**	**2.89**
Defect analysis & trends used to identify process changes.	1.00	1.47	1.96
Number of defects found after release is measured.	1.62	**2.62**	**3.00**
Number of new defects introduced per "fix" is recorded.	1.33	1.87	**2.91**
Time to identify & correct defects is measured.	1.50	**2.30**	**2.91**
Test procedures & policies are clearly identified & described.	1.86	**2.28**	**2.85**

Boldface print indicates common practice (>2.25)

Score	Usage
< 1.25	Uncharacteristic
1.25-1.75	Scattered
1.75-2.25	Significant
>2.25	Accepted

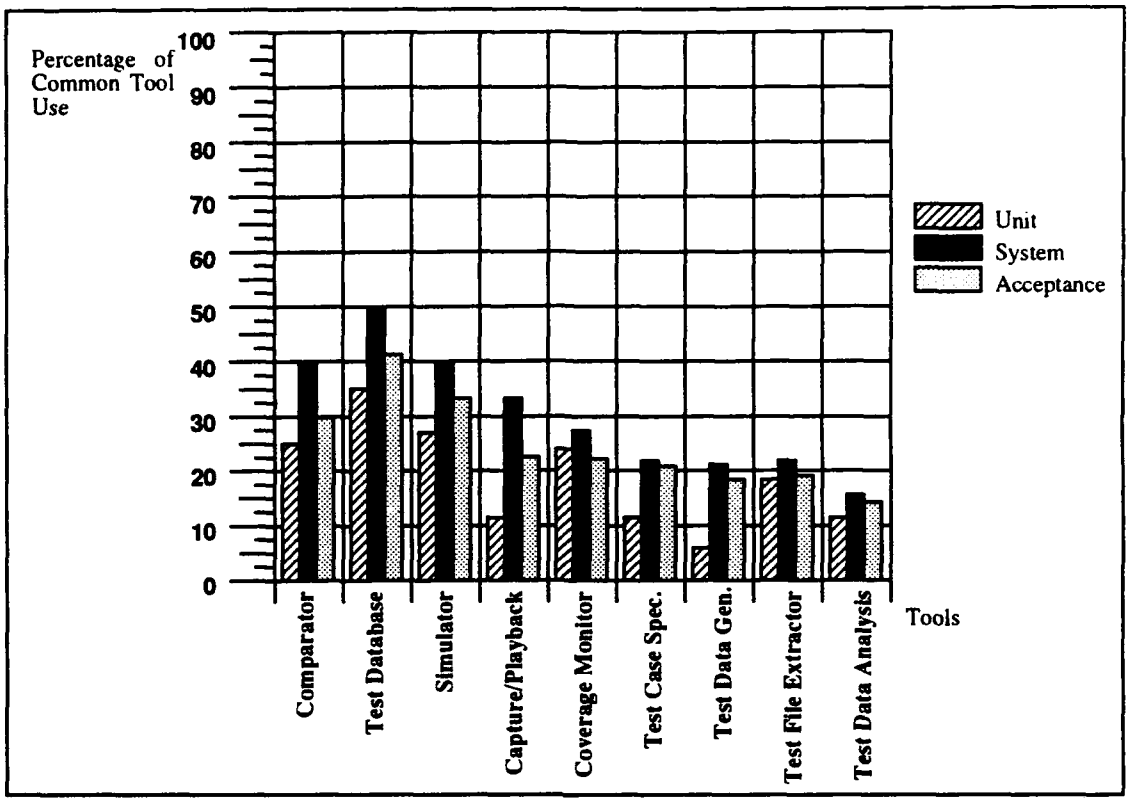


Figure 2-1. Tool Usage Reported in Software Test Practices Survey

Although the results from only two surveys are cited here, there is much data to support these findings. Similar data is provided in, for example, the survey sponsored by the Massachusetts Computer Software Council [KPMG 1992].

3. TEST REQUIREMENTS AFFECTING TOOL USE

This section considers three major drivers that encourage the use of testing tools for the development of SDI software. The first of these is the current set of SDIO documents that provides policy and guidance for software development in general. The second driver is the Software Engineering Institute (SEI) Process Maturity Model (PMM) [Humphrey 1987] that routinely will be used in the near future to conduct evaluations of SDI contractors' software engineering practices. The final driver considered is the Global Protection Against Limited Strikes (GPALS) Computer Resources Working Group (CRWG) software metrics evaluation program.

SDIO requirements are not the only reason to use automated test tools. Indeed, because of the complexity of detail involved in testing even the simplest program, tools are a prerequisite for most forms of static and dynamic analysis. Similarly, the ability to capture, analyze, and present quantitative process measurement data in a meaningful form greatly facilities test management. Although there is a lack of consistent data on the cost effectiveness of particular testing tools, there can be no doubt that automated tools are able to improve the cost effectiveness of testing. One test practitioner, for example, cites reductions in testing time of up to 70%, a 30% increase in overall software development productivity [Graham 1991].

3.1 Affect of SDIO Software Test Requirements on Tool Use

SDIO encourages the use of automated tools to support testing. Candidate tool classes identified in the *Global Protection Against Limited Strikes (GPALS) Software Standards* [GPALS 1992c], for example, are test case generators, performance analyzers, complexity analyzers, and regression analysis tools. The use of source code standards checking, formal verification, and static and dynamic code analysis tools is also discussed. The Trusted Software Guide annex to the GPALS Software Standards requires the use of an automated testbed for creating, executing, documenting, managing, and analyzing the completeness of all tests, and for maintaining test documentation. The *GPALS Software Quality Program Plan* [GPALS 1992a] requires the use of automated metrics data collection and reporting tools.

Additionally, the *SDIO Software Policy* [SDIO 1992b] and the *SDIO Contract Requirements Packages (CRPs) Guidelines for Computer Resource Issues* [GPALS 1992b] impose requirements on testing practices that, either directly or indirectly, foster tool use. These special requirements and their sources are identified in Table 3-1. This table also

Table 3-1. SDIO Test Requirements

TEST REQUIREMENT	SDIO Software Policy	GPALS Sw. Standards	GPALS Sw. Trust Guide	GPALS SQPP	GPALS CRPs	POSSIBLE TOOL SUPPORT
Continuous process improvement. Use of concurrent engineering practices to provide continuous improvement in software engineering processes and the visible quality of products.	√				√	Problem reporting, reliability analysis, cost analysis, progress monitoring
Quality evaluation. Data collection and reporting of a minimum set of software process, product, and management metrics.	√	√		√	√	Quality analysis
Minimum structural test coverage. i) Structural test coverage for CSU/CSCI and regression testing of all statement, branches, loops. ii) Structural coverage and boundary value testing at the unit level, demonstration of coverage at integration level.	√	 √	√ √		√	Structural coverage analysis Structural coverage analysis
Test traceability. Traceability of requirements, design, and code to tests and test results.		√	√		√	Requirements tracking, test planning
Design and code inspections. Formal inspections for all software designs and code products.		√			√	Browsing
Review. Review of CSU tests and results.		√	√			Progress monitoring
Testable requirements. Demonstration of an objective and feasible test of whether each requirement is met.					√	Requirements tracking, test planning
Functional testing. The process of exercising a system under operational conditions to determine that specified functional requirements are implemented correctly.		√	√			Test data & testbed generation, functional coverage analysis
Reliability measurement. Statistical techniques used to reduce observed software defects to acceptable limits.			√			Problem reporting, reliability analysis
Random testing. In addition to other methods for generating test input, random input generated to overcome any test bias.			√			Test data & testbed generation
Penetration testing. Penetration tests required as part of establishing software trust.			√			--
Regression testing. Retest modified software to verify that changes have not caused unintended effects and software still meets the requirements.		√	√		√	Regression & change analysis, requirements tracking, test planning
Test progress tracking. Progress tracked and compared to the Software Test Plan.			√			Test planning, cost analysis, progress monitoring, problem reporting, requirements tracking
Static and dynamic code analysis. Complexity, structure, and style assessment, and checking for language violations, unused code or data.			√			Static and dynamic code analysis, test data & testbed generation
Source code standards compliance. Code portability and style assessment.		√				Auditing, complexity analysis, structure analysis
Test repeatability. The ability to repeat a test with the same inputs and operating conditions to yield the same results.		√	√			Test planning & documentation, testbed generation

identifies the types of testing tools that can be expected to increase significantly the cost effectiveness of the associated test activities.

3.2 Affect of the SEI Process Maturity Model on Tool Use

Starting in FY93, SDIO will require evaluation of contractor software engineering capabilities using the SEI PMM. This evaluation will be routinely conducted as part of source selection activities, and yearly during the course of a contract, by an independent team of evaluators. Contractors will be encouraged to perform annual self-appraisals. The PMM is used to rank software engineering capabilities as the following:

- Level 1 - Initial. The software process is characterized as *ad hoc*, and occasionally even chaotic. Few processes are defined and success depends on individual effort.
- Level 2 - Repeatable. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
- Level 3 - Defined. The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software.
- Level 4 - Managed. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures.
- Level 5 - Optimized. Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies.

As part of the evaluation, the PMM queries the use of automated requirements trackers, test data generators, coverage analyzers, complexity analyzers, cross-referencers, and interactive source-level debuggers. The testing-related questions that are asked in determining the software engineering capability level are listed in Table 3-2. This table identifies the types of testing tools that could be used to support the identified activities. The PMM also addresses the use of process and product measures for monitoring the status and quality of both the developing product and the development process. In this case, data collected in the course of testing activities can serve several purposes. In addition to supporting the determination of the effectiveness of actual testing activities, this data provides valuable insight into other development activities such as defect prevention, training, and software quality assurance. Test-related data can also be used in the assessment of the benefits and effec-

tiveness of existing and new technology. Test tools can support the collection of much of this data.

The PMM is currently being revised. The new version, called the Capability Maturity Model [Paulk 1991], extends the information sought on testing practices, such as the following:

Table 3-2. PMM-Implied Test Requirements

KEY AREA	PMM QUESTIONS	Level	POSSIBLE TOOL SUPPORT
Process Metrics	Are statistics on software code and errors gathered?	2	Problem reporting, static analysis
	Are code and test errors projected and compared to actuals?	4	Problem reporting, test planning
	Are profiles maintained of actual versus planned software units completing unit testing over time?	2	Progress monitoring, test planning
	Are profiles maintained of actual versus planned software units integrated over time?	2	Progress monitoring, test planning
	Is test coverage measured and recorded for each phase of functional testing?	4	Coverage analysis
	Are software trouble reports resulting from testing tracked to closure?	2	Problem reporting
	Is test progress tracked by deliverable software component and compared to the plan?	2	Progress monitoring, test planning, cost analysis
Data Management & Analysis	Is error data from code reviews and tests analyzed to determine the likely distribution and characteristics of the errors remaining in the product?	4	Problem reporting
	Are analyses of errors conducted to determine their process-related causes?	4	Problem reporting
	Is a mechanism used for error cause analysis?	5	--
	Is software productivity analyzed for major process steps?	4	Progress monitoring
Process control	Is there a mechanism for assuring that regression testing is routinely performed?	2	--
	Is there a mechanism for ensuring the adequacy of regression testing?	3	Change analysis, coverage analysis
	Are formal test case reviews conducted?	3	
Documented Standards & Procedures	Are standards applied to the preparation of unit test cases?	3	DoD document generation
	Are coding standards applied to each project?	2	Auditing
	Are formal procedures applied to estimating software development schedules/cost?	2	Progress monitoring, test planning, cost analysis

- Verification of software requirements, design, and code according to the project's defined software process.
- Use of formal criteria to determine readiness for any level of testing.
- Review of test plan, test procedures, and test cases by peers of the developers of the plan and procedures before they are considered ready for use.
- Appropriate change of the test plan, test procedures, and test cases whenever the allocated requirements, software requirements, software design, or code being tested changes.
- Determination of the adequacy of testing based on the level of testing performed, the test strategy selected, and the test coverage to be achieved.
- Performance of formal system testing of the software, according to the project's defined software process, to ensure that the software satisfies the software requirements.
- Performance of acceptance testing of the software, according to the project's defined software process and approved acceptance test plan, to demonstrate to the customer and end users that the software satisfies the allocated requirements.
- Maintenance of consistency across the software engineering products, including the software plans, allocated requirements, software requirements specification, software design, code, test plans, and test procedures.

Here again, test tools can be expected to play an important supporting role.

3.3 Affect of the Software Metrics Program on Tool Use

No set of metrics for software project management has gained widespread acceptance by software developers. Accordingly, the GPALS CRWG on Software Quality Improvement and Standards (SQI&S) has developed a Software Metrics Evaluation Plan (SMEP) [SDIO 1992a] designed to evaluate and provide SDIO with recommendations on metrics and metrics tools that can be implemented SDI-wide. This on-going program will involve the evaluation of several sets of metrics and metrics tools on a number of different SDI software development projects. The first evaluation is expected to proceed through 1993 and will be conducted on the SDI Level 2 System Simulator (L2SS).

The SMEP considers three functional classes of metrics: management, process, and product metrics. The metrics chosen for initial evaluation include those identified by the Army's Software Test and Evaluation Panel (STEP) [U.S. Army 1992]. Metrics from the Air Force's Software Management Indicators [AFSCP 1986] and from Martin Marietta's Pro-90 Software Metrics Handbook [Martin Marietta 1991] will be used to estimate com-

puter resource use. Table 3-3 identifies specific SMEP metrics and the types of tools that support their evaluation.

Table 3-3. Software Metrics Plan Implied Test Requirements

METRIC TYPE	METRIC	POSSIBLE TOOL SUPPORT	CRWG Evaluated Tool Support				
			SASET	SPCR	Analyze	ADAMAT	SQMS
Management	Sizing	Cost modeling	√				
	Costing, Schedule, Manloading	Cost modeling	√				
	Computer Resource Utilization	--					
	Requirements Analysis	Requirements tracking					
Process	Nonconformance Reporting	Problem reporting		√			
	SDP & Software Standards	Auditing					
	Utilization of Software Tools	Tool inventorying					
	Configuration Management	Change control					
	Change Summary Process	Problem reporting					
	Productivity Measures	Progress monitoring			√	√	
	Development Progress	Progress monitoring					
	Cost	Cost analysis					
Product	Defect Density	Problem reporting		√			
	Maintainability	Quality analysis			√	√	
	Cyclomatic Complexity	Complexity analysis			√	√	√
	I/O Statements	Quality analysis, static analysis			√		
	Entry & Exit Points	Quality analysis, static analysis			√		
	Volume	Complexity analysis			√		√
	Portability	Quality analysis				√	
	Reliability	Reliability analysis				√	√
	Documentation	Document generation					

To date, the CRWG has sponsored the examination of the following five tools to assess their support for the application of the SMEP metrics in the L2SS evaluation [Martin Marietta 1992]:

- Software Architecture, Sizing, and Estimating Tool (SASET). A cost, schedule, and sizing model that provides software development estimates.
- Software Problem and Change Report (SPCR). Tracks and reports on nonconforming conditions and the status of closure and corrective actions.

- Analyze. Estimates productivity in terms of the ratio of the number of executable lines of code to the total lines of source code and collects statistics on source code.
- Ada Measurement and Analysis Tool (ADAMAT). Collects some 150 parameters to estimate software reliability, maintainability, and portability.
- Software Quality Management System (SQMS). Collects parameters to estimate software reliability, complexity, and a quality index.

4. APPROACH AND METHODS

The overall approach taken to this work was to identify suppliers of testing tools, select tools for examination, and apply the selected tools in the testing of sample pieces of code. The tools examined to date are all available independently of any particular computer-aided software engineering (CASE) system or Ada development environment. This section also summarizes the types of testing support provided by these larger-scale products so that their testing capabilities can be contrasted with those provided by the independent tools.

4.1 Tool Selection

Nearly four hundred suppliers of over six hundred tools were identified. From this initial set of suppliers, a short list was prepared of those tools that support static and dynamic analysis of Ada code, test management, and problem reporting. Information was sought from the appropriate suppliers. In several cases, suppliers gave in-house demonstrations of their tools. Additional criteria were then applied to refine the short list to be compatible with the resources available for tool examination. To ensure that the results apply to the largest possible audience, it was decided that selected tools should be essentially independent of processor architecture. Consequently, tools such as non-intrusive coverage monitors which require special purpose hardware were not considered.

Tool selection also considered work performed by other groups. The GPALS CRWG has examined and reported on five related tools. Most of these tools are available on VAX or Sun platforms. The Air Force Software Technology Support Center (STSC) has reported on several categories of software tools, testing tools being one of these categories [Sittenauer 1991]. The role of the STSC is to assist Air Force Software Development and Support Activities in the selection of technologies that improve the quality of Air Force software products and increase the productivity of its efforts; the focus is on the long-term development and support of Mission Critical Computer Resources (MCCR) software. STSC looked at test tools that support Ada, assembler, ATLAS, C, Fortran, and Jovial programming languages running on DEC/VAX equipment, HP/Apollo and Sun workstations, or IBM and Macintosh personal computers (PCs). The STSC 1991 report provides half-page descriptions of some twenty eight tools, and tool critiques based on hands-on application for eight of these tools.[2] Table 4-1 identifies the tools examined in the CRWG and STSC studies. Care was taken not to duplicate this previous work.

Table 4-1. Tools Examined in the CRWG and STSC Studies

STUDY	TOOL NAME	TOOL SUPPLIER	LANGUAGES SUPPORTED					TEST CAPABILITIES					
			Ada	C	C++	Fortran	Other	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis	Regression Analysis	Cost Modeling
STSC	Automator qa	Direct Technology	+	+	+	+	+					√	
	AutoTester	Software Recording Corporation	+	+	+	+	+					√	
	Bloodhound	Goldbrick Software	+	+	+	+	+					√	
	Logiscope	Verilog, Inc.	√	√		√	√			√	√		
	PC Metric	SET Laboratories, Inc.	√	√		√	√			√			
	VAX PCA	Digital Equipment Corp.	*	*	*	*	*				√		
	VAX SCA	Digital Equipment Corp.	*	*	*	*	*			√			
	Test Manager	Digital Equipment Corp.	+	+	+	+	+					√	
CRWG	ADAMAT	Dynamics Research Corp.	√							√			
	Analyze	Martin Marietta IS		√		√				√			
	SASET	Martin Marietta IS	+	+	+	+	+						√
	SQMS	Martin Marietta IS	√	√		√	√	√		√			
	SP/CE	Martin Marietta IS	+	+	+	+	+		√				

\+ - Language independent
* - Most VAX supported languages

Table 4-2 identifies the tools already examined in the IDA study and Table 4-3 identifies several additional tools awaiting examination as part of this ongoing work. In most cases, this latter group are new tools due to be released late in 1992 or early in 1993. Some offer unique capabilities that fill identified gaps in testing tool functionality. PARTAMOS, for example, is expected to provide for reproducible testing of concurrent Ada software. Others provide capabilities that are, as yet, not commonly available. For example, Ada-ASSURED and the Ada Quality Toolset will check for conformance of code with the Software Productivity Consortium (SPC) Ada style guidelines [SPC 1991] selected by SDIO. The U.S. Government is sponsoring development of ARC SADCA, and NATO the development of the Test Support Toolset of the NATO Ada Programming Support Environment

2. The 1992 update of this report, divided into two reports *Test Preparation, Execution, and Analysis Tools Report* [Price 1992a] and *Source Code Static Analysis Test Tools Report* [Price 1992b], does not include any tool critiques.

Table 4-2.Tools Examined in the IDA Study

TOOL NAME	TOOL SUPPLIER	LANGUAGES SUPPORTED					TEST CAPABILITIES				
		Ada	C	C++	Fortran	Other	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis	Regression Analysis
ADADL Processor	Software Systems Design	√	√		√				√		
AdaQuest	General Research Corp.	√							√	√	
AutoFlow-Ada	AutoCASE Technology	√	√			√			√		
DDTs	QualTrak Corp.	+	+	+	+	+		√			
EDSA	Array Systems Computing, Inc.	√							√		
GrafBrowse	Software Systems Design	√	√		√				√		
LDRA Testbed	Program Analysers, Ltd.	√	√	√	√	√	F		√	√	
Logiscope	Verilog, Inc.	√	√	√	√	√			√	√	
MALPAS	TA Consultancy Services, Ltd.	√	√		√	√			√		
Metrics Manager	Computer Power Group, Inc.	+	+	+	+	+	√				
QES/Manager	Quality Engineering Software, Inc.	+	+	+	+	+	√				
QualGen	Software Systems Design	√	√		√				√		
S-TCAT	Software Research, Inc.	√	√		√	√			√	√	
SQA:Manager	Software Quality Automation	+	+	+	+	+	√	√			
SRE Toolkit	Software Quality Engineering	+	+	+	+	+	√				
SoftTest	Bender & Associates	+	+	+	+	+				√	
T	Programming Environments, Inc.	+	+	+	+	+				√	
T-PLAN	Software Quality Assurance, Ltd.	+	+	+	+	+	√	√			
TBGEN	Testwell Oy	√	√	√						√	
TCAT	Software Research, Inc.	√	√	√	√	√			√	√	
TCAT-PATH	Software Research, Inc.	√	√		√	√			√	√	
TCMON	Testwell Oy	√	√	√						√	
TDGen	Software Research, Inc.	+	+	+	+	+				√	
TSCOPE	Software Research, Inc.	+	+	+	+	+				√	
TST	STARS Foundation Repository	√							√	√	
Test/Cycle	Computer Power Group, Inc.	+	+	+	+	+	√	√			
TestGen	Software Systems Design	√	√						√	√	

+ - Language independent
F - Future capability

(APSE). Both of these toolsets are expected to provide a broad range of static and dynamic testing capabilities.

Table 4-3. Tools Planned for Future Examination

TOOL NAME	TOOL DEVELOPER/ SUPPLIER	LANGUAGES SUPPORTED					TEST CAPABILITIES				
		Ada	C	C++	Fortran	Other	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis	Regression Analysis
ARC SADCA	Optimization Technology, Inc.	√	√			√			√	√	
Ada-ASSURED	GrammaTech, Inc.	√							√		
Ada Quality Toolset	Marlstone Software Technology, Inc.	√							√		
Battlemap Analysis Tool	McCabe & Associates	√	√		√	√			√	√	
CaseQMS	Advisor & Computer Systems, Inc.	+	+	+	+	+		√			
Instrumentation Tool for Ada	McCabe & Associates	√								√	
PARTAMOS	Alcatel Austria	√		√					√	√	
QES/Architect	Quality Engineering Software, Inc.	+	+	+	+	+	√				√
QES/Programmer	Quality Engineering Software, Inc.	+	+	+	+	+				√	
QTET	QualTrak Corp.	+	+	+	+	+	√				√
QUES	Software Productivity Solutions, Inc.	√							√		
QualityTEAM	Scopus Technologies	+	+	+	+	+		√			
Requirements Tracer	Teledyne Brown Engineering	+	+	+	+	+	√				
SLICE	McCabe & Associates	√	√		√	√			√	√	
START	McCabe & Associates	+	+	+	+	+				√	
SQMS	Software Quality Tools Corp.	+	+	+	+	+	√	√			
SWEEP	Software Productivity Consortium	+	+	+	+	+	√				

+ - Language independent

4.2 Method of Examination

Each static and dynamic analysis tool was used to test several small Ada programs. The goal of these initial tool applications was to allow the examiner to gain familiarity with overall tool operation. Each tool was subsequently applied to the same Ada program. This software was the Ada Lexical Analyzer Generator program that creates a lexical analyzer or "next-token" procedure for use in a compiler or other language processing program [Meeson 1989]. It was developed for the Software Technology for Adaptable, Reliable Systems (STARS) program and consists of several Ada subprograms with a total of over three thousand lines of code. In the absence of a historical test database, the test management and problem reporting tools were examined using the sample test database provided by each supplier.

Generally, suppliers provided their latest tool release for the examination. In a couple of cases, only demonstration versions were available. In each such case, however, the demonstration version was fully functional and only limited by the number of inputs it could accept.

4.3 CASE System Support for Software Testing

A recent survey of CASE vendors, performed on behalf of the U.S. Air Force, found that nearly 25% of the examined products claim explicit support for software testing activities [Hook 1991]. The goal of incorporating testing support into a CASE system is to provide easy access to testing tools and so facilitate continual evaluation of evolving software. This evaluation can be used to ensure timely detection of faults and provide the software developer with feedback to guide the development process, thus encouraging a better integration of testing with other software development activities.

Table 4-4 indicates the types of test support provided by current CASE systems. At the code level, coverage and performance analysis are the most common types of support provided. These capabilities are similar to those provided by independent test tools and are sometimes available as stand-alone products. However, it is during earlier stages of software development that CASE systems hold the most potential for improving the integration of testing with other development activities. Several CASE tools provide requirements traceability, use simulation and, occasionally, executable specifications to indirectly support testing. Recently, more direct support in terms of test generation, test plan tracking, and specification analysis based on user-defined rules has become available. Examination of testing tools that are part of a CASE system is still needed. In particular, the question of how to achieve the necessary integration of independent and CASE-based testing tools to provide a comprehensive automated test capability must be addressed.

4.4 Development Environment Support for Software Testing

A previous IDA study identified twenty eight U.S. companies that supply validated Ada compilers [Hook 1991]. All these vendors provide a minimum set of tools for Ada code development including the compiler, editor, debugger, library manager, and run-time environment. The Ada language definition allows Ada compilers to provide considerably more static analysis than is possible for older languages such as Fortran. Capabilities such as type

checking and range checking, for example, are always provided. The other types of testing support provided vary quite considerably. As shown in Table 4-5, coverage analysis, performance analysis, and cross-referencing are the most common testing capabilities supported.

Some vendors (DEC, IBM, and Verdix) demonstrate a movement towards providing an integrated development environment that encompasses most phases of the software development life cycle. In this case, for the implementation phase, there are tool sets offered with the compiler. For requirements specification and design, these development environments support various off-the-shelf CASE systems.

Table 4-4. CASE-based Testing Support

CASE NAME	VENDOR	SPECIFICATION			DESIGN		CODE							
		Reqs. Traceability	Simulation	Support Type	Simulation	Support Type	Complexity Analysis	Coverage Analysis	Performance Analysis	Quality Analysis	Test Data Generation	Other	Problem Reporting	Change Reporting
ASA/AGE (with Logiscope)	Verilog USA	√	√	Static verification, test case generation, complexity analysis	√	Dynamic checking, test process generation	*	*		*				
AISLE (with TestGen)	Software Systems Design, Inc.	√				Expert review assistant	*	*		√				
APJ Workbench Animator	Michael Jackson Systems Ltd.										√			
Auto-G	RJO Enterprises, Inc.		√											
CAEDE	Carleton University								√					
CASE Station	Mentor Graphics, Inc.			Checking by user-defined rules				√						
Classic Ada	SPS								√			√		
Design Generator	CSC					Complexity analysis								
Envision	Future Technology, Inc.	√	√	Exec. spec. & test data generation										
Foresight	Athena Systems, Inc.		√	Exec. spec. with tracing										
HP Workbench	Hewlett-Packard							√	√			√		
MODEL, NETworkbench	CCC Co.										√			
Maestro	Softlan, Inc.			Inferencing by user-defined rules									√	√
ProMod	ProMod, Inc.	√		Test plan tracking										
REFINE	Reasoning Systems	√		Exec. spec. with assertions							√	√		
re/NuSys Workbench	Scandura Intelligent Systems				√	Exec. spec. & coverage								
SES Workbench	SES	√			√	Exec. spec. with assertions & perf. analysis								
Statemate	i-Logix, Inc.			Exec. spec. & test data generation for emulation	√									√
StP	IDE	√				Parameter checking								
System Engineer/Architect	Popkin Software	√						√						
TAGS/CASE2	Teledyne Brown Engineering	√	√				√							
Teamwork	Cadre Technologies, Inc.	√	√				√	√	√		a			
VSF family	Systematica Ltd.			Checking by user-defined rules									√	
001	Hamilton Technologies, Inc.		√	Exec. spec. with timing constraints							√			

* - Reviewed as independent test tool
a - Via the T Testing tool

Table 4-5. Ada Development Environment-based Testing Support

VENDOR	DEVELOPMENT ENVIRONMENT & OTHER (TEST) PRODUCTS	Simulation	Coverage Analysis	Code Size Analysis	Performance Analysis	Verification Support	Cross-Reference Analysis	Pretty Printing	Browsing	Profiling	Compilation Order Analysis	Regression Analysis	Dependency Analysis	Debugging	Test Manager	Problem Reporting
AETECH	Ada Software Development Toolset, AdaScope													√		
Alliant Computer Systems	FX/Ada Development System							√						√		
Alsys, Inc.	Ada Software Development Environment		√		√		√									
	AdaProbe, AdaTune, AdaXref, AdaFormat				√		√							√		
Digital Equipment Corp.	VAXset		√		√										√	
	DEC FUSE						√		√	√				√		
DDC International	DACS Tool Set family		√	√		√					√			√		
E-Systems, Inc.	Tolerant Ada Development System													√		
Harris	Harris Ada Programming Support Environment									√				√		
Hewlett Packard	AxAda Programming Software Environment		√		√											
	HP Ada/SoftBench													√		
IBM	IBM Ada/370 family						√	√		√			√	√		
	AIX Ada/6000													√		
Intermetrics	Ada Development Environment, AdaView													√		
Irvine Compiler Corp.	ICC Ada Software Development & Test Environment		√							√				√		
MIPS Computer Systems	Ada Development Environment													√		
Meridian Software Systems	Open Ada, Meridian Ada													√		
R.R. Software	Ada Development Environment													√		
Rational	Rational Environment family						√							√		
	TestMate		√						√			√			√	
SD_SCICON	Ada Development Environment	√						√						√		
Sun Microsystems	Sun Ada Development System		√													√
Tartan Laboratories, Inc.	Ada VMS Compilation System family	√					√							√		
TeleSoft	TeleArcs						√		√				√	√		
	TeleGen2 family						√	√			√		√	√		
Verdix Corp.	Sun Ada Development Environment, VADS Workstation													√		
	Statistical Profiler									√						

5. TEST MANAGEMENT

This section identifies key capabilities of the examined tools in terms of the support provided for test management. It is intended to provide a quick overview of the types of automated support available in each area and insight into how this support can be used to facilitate software testing.

Previously Table 4-2 identified six tools as providing test management capabilities and one other, LDRA Testbed, as currently being extended to provide these capabilities in the near future. The functionality of these tools is further detailed in Table 5-1.

Table 5-1. Test Management Capabilities of Examined Tools

TOOL NAME	Test Planning & Documentation												Reqs Map		Progress Monitoring					
	Test Plan Capture	Test Case Capture	Test Log	Test Inventory Software	Test Inventory Tools	User-def. Data Entry Templates	Cross-Referencing Test Items	Version Control	IEEE Standards Support	DoD/Mil Standards Support	User-defined Reports	Predefined Reports	Requirements/Test Tracing	Change Analysis	Test Case Status Reporting	Requirements Coverage	Test Schedule Reporting	Cost Reporting	Reliability Analysis	Productivity Analysis
LDRA Testbed																			F	
Metrics Manager																				√
QES/Manager		√			√	√					√									
SQA:Manager	√	√	√	√	√	√	√	√	√	√	√	√			√			√	√	
SRE Toolkit																		√	√	
T-PLAN	√	√	√	√		√	√	√			√	√	√	√	√		√			
Test/Cycle	√	√	√	√		√	√	√			F	√	√	√	√	√		√	F	

F - Future capability

Two additional tools, SoftTest and T, are also discussed. Although not properly classed as test management tools, both of these provide some support for requirements mapping and progress monitoring.

5.1 Test Planning and Documentation

Test planning is a prerequisite to effective management of test activities. It provides the base against which required test activities can be scheduled, test resources can be estimated, and the progress of test activities can be tracked.

Of the examined tools, QES/Manager, SQA:Manager, T-PLAN, and Test/Cycle provide explicit support for test planning, though they take somewhat different approaches. QES/Manager and SQA:Manager incorporate a predefined test model that defines the relationship among test objects such as documents, test cases, and products. In the case of SQA:Manager this model follows the Institute for Electrical and Electronics Engineers (IEEE) standard test model [ANSI/IEEE 1983]. The QES/Manager test model groups test cases into test drivers that specify an execution sequence for those test cases. Test/Cycle defines the types of permissible test objects, but allows the user to define the links between these. It is worth noting that software builds are one of Test/Cycles object types, allowing this tool to explicitly support incremental software development. T-PLAN provides the most flexibility. It requires a user to start by defining the underlying test model, although an in-house developed test methodology can be used as the source of the test model if desired.

With the necessary model established, these tools capture similar information for test cases and groupings of these test cases. They differ in the other types of information captured. Most significantly, only T-PLAN and Test/Cycle explicitly capture requirements and trace these to testing data (see Section 5.2), and only SQA:Manager, T-PLAN, and Test/Cycle explicitly document a test plan. All the tools except QES/Manager do, however, trace test data to the software items under test. (A capability offered by QES/Manager, unique among these tools, is the ability to simulate the test data.) Examples of other information that can be captured by some of these tools include a test schedule and an inventory of test tools. All these tools provide user-tailorable templates to support data entry.

The tools also differ in their reporting on the contents of the test library. QES/Manager requires the user to define all report formats, and Test/Cycle provides a range of predefined report formats. SQA:Manager and T-PLAN support both predefined and user-defined report formats. In addition to the available IEEE standards, SQA:Manager supports applicable DoD standards DoD-STD-2167A and DoD-STD-2168, and military standard MIL-STD-480.

5.2 Requirements Mapping

The ability to trace the relationship between software requirements and test items provides valuable insight into the completeness and effectiveness of both test planning and test execution. It is also a prerequisite for the change analysis that determines the potential scope of effect of a proposed requirements change. The ability to provide this support is one of the major differences between the tools in this category. It is available with T-PLAN and Test/Cycle.

Test/Cycle uses requirements validation matrices to cross-reference requirements against software builds, test runs, and test cases. These matrices can be examined to ensure that all requirements are appropriately covered, providing quick insight into test planning completeness. T-PLAN links requirements to test cases via test conditions that can be grouped to reflect, for example, valid/invalid categories, system releases or versions. It also reports on the test items affected by a change to a test requirement, in addition to the change analysis provided for other types of test items.

SoftTest and T provide requirements traceability in a different way. Here a requirements specification is used to guide the generation of test cases. Hence, test cases are automatically linked to defined functional requirements. Both tools provide matrices that give a quick visual guide to the cross-referencing between functional requirements and test cases.

5.3 Test Progress Monitoring

Test progress monitoring is important for effective management of test activities. By tracking actual progress against planned progress, managers can get an early indication of potential schedule slips to support timely decision making. The early identification of quality shortfalls is another piece of valuable information. The data collected during test progress monitoring can also be used to assess various overall software development indicators and quality indicators (see, for example, [AFSCP 86, AFSCP 87]). Progress monitoring is largely based on a log of testing activities. Data is entered into the test log manually or, in some cases, can be imported from a test execution tool.

SQA:Manager and T-PLAN capture similar information for the test log. Using this information, SQA:Manager reports on the status of each test case, that is, the number of tests passed, failed, and aborted, and the number of incidents raised. T-PLAN reports whether

each test procedure has been tested, date of last test, and whether a re-test is required, together with details on the conduct of the individual tests performed. Using the schedule information entered for each test specification, T-PLAN also compares estimated and actual levels of effort to determine the outstanding effort and report on the percentage completion. This reporting is available for test planning, testing, regression testing and review activities.

Test/Cycle reports on the validation status of requirements, builds, and test runs in terms of the percentage of test cases passed. It provides this for each leaf requirement or requirement subtree in its requirements hierarchy. Additional reports summarize the overall status of requirements, builds, and test cases, whereas a test log report provides detailed information on the status of individual test cases.

SoftTest and T report on the requirements coverage achieved through testing to date. SoftTest reports requirements coverage in terms of the number of functional variations tested with respect to those testable; this requires the user to manually enter the results of test case execution. T also maps user-supplied test results to requirements to report on test adequacy with respect to requirements coverage. It provides a test comprehensiveness measure that, at the user's choice, combines requirements, input/output, and structural coverage.

Reliability analysis is also used to monitor test progress against a stated objective. A failure intensity objective, for example, specifies the expected number of software failures per unit of time, whereas a reliability objective specifies the probability of failure-free operation. By looking at the occurrence of software failures during testing activities, it is possible to estimate the number of defects remaining in a piece of software and determine (with confidence intervals) the additional time or resources needed to reach the goal objective. By predicting the reliability of software after modification, these measures can also help to time the performance of maintenance activities, for example, the addition of new features. Under the proper conditions, reliability measures can be used to help determine the effectiveness of particular software engineering practices, or the effects of process improvements.

Many different reliability models have been proposed. The two most common are Musa's basic execution-time model and the Musa-Okumoto logarithmic Poisson execution-time model [Musa 1987]. Both models characterizes failures as a nonhomogeneous Poisson distribution. SRE Toolkit supports reliability analysis using both of Musa's models, whereas SQA:Manager uses the Musa-Okumoto model. Both tools provide failure intensity and reliability reports that include the amount of additional testing time needed to meet a tar-

geted reliability, and an estimation of how many more problems are likely to be found during that additional testing. They both support cost analysis. SQA:Manager relates the hours spent in test activities and in problem resolution to cost rates in a cost base to report the cost of these activities. Using data on the cost of failure identification and correction, and the cost of operational failure, SRE Toolkit maps total life cycle, system test, and operational life costs against a specified failure intensity objective.

SRE Toolkit supports a number of additional features. For example, the user can specify a failure time adjustment to take account of incremental delivery of software to the system test process and a testing compression factor to specify the ratio of field to test execution time. The toolkit can be instructed to interpret individual failure entries as independent failure events or to perform grouped data analysis. A suite of prototype programs provides further information such as summary statistics for each recording period, estimates of resource usage calendar time parameters from resource usage data, and plots of completion date for testing and life cycle costs versus failure intensity objective.

In addition to that discussed here, information on the status of identified problems (see Section 6) and the coverage achieved during dynamic testing (see Section 8.2) also provide insight into the status of testing activities.

5.4 Productivity Analysis

Productivity data, like quality data, can be used to monitor the efficiency of the software development process. It supports the identification of those instances where process improvements are needed, and the effectiveness of process changes. While several tools support the collection and analysis of quality data, Metrics Manager is the only examined tool that provides productivity analysis. As such, it looks at a user-defined Management Information System (MIS) function, collecting data on a monthly, quarterly, or annual basis to monitor the performance of the organization and track the impact of new methods, organizational structures, and technologies. Metrics Manager is supported by an industry database that allows comparison of organizational data against industry statistics.

6. PROBLEM REPORTING

The primary purpose behind problem reporting is to ensure that all identified problems are addressed. The data inherent in this activity serves several additional purposes. It provides a valuable insight into both the software status and the progress of development and test activities. Additionally, it provides much of the data needed to drive continuous process improvement activities.

Four tools that support problem reporting were examined. One of these, DDTs, addresses this function exclusively. For SQA:Manager, T-PLAN, and Test/Cycle, problem reporting is only one of the types of support provided for software testing. Consequently, it is not surprising that there are several significant differences between these two types of support. The capabilities of the tools are summarized in Table 6-1.

Table 6-1. Problem Reporting Capabilities of Examined Tools

TOOL NAME	Report Types			Details Captured								Import Capability	Reporting				Stnd. Con.		Distrib. Archit.		
	Defect Reports	Change Requests	Incident Reports	Define Categories	Set Status	Assign Resolution	Set Resolution	Set Effort	# Priority/Severity Levels	Set Phase Introduced	Set Changed Software		Predefined Reports	User-defined Reports	Report Filters	Query/Search Capability	IEEE Standards	DoD Standards	Automatic Notification	Multiple Projects	Administrative Functions
DDTs	√	√	√	√	√	√	√	√	5	√	√	√	√	√	√	√	√	√	√	√	√
SQA:Manager	√		√	√	√	√	√	√	5	√	√	√		√	√						
T-PLAN			√	√			√	√	3				√								
Test/Cycle	√	√	√	√	√			√	3	√			√								

6.1 Report Types and Details Captured

The ability to distinguish among different types of problems, and perform separate tracking and reporting for each type, is very useful in monitoring the software development progress and planning further development activities. The common types of problem reports are incident reports, defect reports, and change requests. Only Test/Cycle has all these problem types, collectively called *work requests*, built in. Although in its basic form DDTs

only distinguishes between defects and change requests, it can be customized to also accept incident reports. SQA:Manager distinguishes between incidents and defects. T-PLAN tracks and reports a single problem type, called *service queries*.

By and large, all the tools capture similar details about identified problems. Data entry is guided by user-tailorable templates. DDTs allows the provision of supplemental information that is kept separately, but linked to a defect report. This additional information can be used, for example, to include the data files needed to reproduce a problem. The test item(s) to which problem reports are linked affects the type of tracking that can be performed. SQA:Manager and T-PLAN link them to, respectively, test cases and test specifications. DDTs and Test/Cycle link problem reports to software items.

DDTs provides a good example of the additional power provided by tools that focus exclusively on problem reporting. Here problems have a specified life cycle defined in terms of states and state transitions. The system administrator is permitted to modify this life cycle.

6.2 Import Capability

A flexible import facility is a valuable feature. It allows data generated using other tools to be incorporated in a common problem database. This is useful, for example, when different problem reporting tools are used, perhaps to cater for different development organizations or host machines. DDTs and SQA:Manager provide this capability.

6.3 Reporting Capabilities

T-PLAN, Test/Cycle, and DDTs provide predefined report formats. In the case of DDTs, these reports conform to DoD-STD-2167A and the proposed IEEE standard classification for software errors, faults, and failures [IEEE 1987]. For T-PLAN the available statistical reports analyze the total numbers of defects, or queries, by classification. Frequency of defects and defect resolution is also provided, as well as the percentage complete and outstanding effort required to complete approved changes. Test/Cycle reports provide only work request descriptions and a work request log. DDTs also allows a user to define his own report formats, as does SQA:Manager. In these cases, a number of predefined report filters and sorting keys are provided to support reporting based on any problem characteristic.

SQA:Manager and Test/Cycle report on the costs associated with defect detection and correction. Of the examined tools, only DDTs provides a capability for automatic weekly reporting on problems.

In addition to its reporting facilities, DDTs provides advanced search and query functions.

6.4 Standards Conformance

SDIO requires the reporting and tracking of identified problems but does not specify how this requirement should be met. Some additional guidance is given in DoD-STD-2167A. This standard requires, for example, that problems are classified by category (software, documentation, or design problem) and are assigned one of five levels of priority. It also requires analysis of defect trends and the identification of any additional problems introduced by a problem fix.

The examined tools vary in their ability to meet these requirements. As shown in Table 6-1, only DDTs and SQA:Manager provide five priority levels as a default option, although, for the other tools, the user can generally modify the input template to allow a different set of levels. The ability to record problem classifications is highly variable. SQA:Manager and T-PLAN, for example, allow user-defined categories, whereas DDTs accepts free-form text for this information. None of the tools provides explicit support for recording the introduction of new problems as a results of a problem fix. Several pieces of information can support the analysis of defect trends. Problem classification and details on when a problem was inserted, detected, and the first opportunity for its detection, for example, are all useful. DDTs, SQA:Manager, and Test/Cycle capture at least part of this information.

6.5 Distributed Architecture

The dedicated problem reporting tool, DDTs, is network based. This tool uses electronic mail to provide automatic notification of changes in problem status and to support remote problem entry. It also supports multiple projects. DDTs also provides access controls and various other administrative capabilities. These additional capabilities range from checking and repairing the database to template definition.

7. STATIC ANALYSIS

Static analysis is used to determine the presence or absence of particular, limited classes of errors, to produce certain kinds of software documentation, and to assess various characteristics of software quality. Unlike dynamic analysis, static analysis can sometimes be performed on incomplete or partly development products and does not necessitate costly test environments. It cannot, however, replace dynamic analysis, although it can be used to guide and focus dynamic testing. Previously Table 4-2 identified fourteen tools as supporting static analysis. The functions provided by these tools are summarized in Table 7-1.

Table 7-1. Static Analysis Capabilities of Examined Tools

TOOL NAME	Complexity Analysis	Data Flow Analysis	Control Flow Analysis					Information Flow Analysis	Standards Conf. Analysis	Quality Analysis	Cross-Reference Analysis	Browsing	Symbolic Evaluation	Specification Compliance	Pretty Printing
			Structure Analysis	Path Analysis	Code Statistics	Flowgraph Generation	Call Graph Generation								
ADADL Processor	√										√				√
AdaQuest				√					F		F				
AutoFlow-Ada						√	F								√
EDSA		√		√							√	√			√
GrafBrowse							√					√			
LDRA Testbed	√	√	√	√	√	√	√	√	√		√				√
Logiscope	√		√	√	√	√	√			√		√			
MALPAS	√	√	√	√		√		√					√	√	
QualGen										√					
S-TCAT					√		√					√			
TCAT					√	√						√			
TCAT-PATH	√			√	√	√						√			
TST				√					√						√
TestGen	√			√		√									

F - Future capability

7.1 Complexity Analysis

Complexity measures are put to various test-related uses. McCabe has developed a method, called Structured Testing, that uses cyclomatic complexity to guide the selection

of a minimum set of required paths to test [McCabe 1982]. Complexity measures are also used to estimate the number of defects present in a piece of software and to identify pieces of code that are potentially defective.

Models for estimating program complexity have been based on various characteristics of software structure and semantics. The best-known set of complexity measures are all applied at the program unit level. They are McCabe's cyclomatic complexity metrics [McCabe 1976] and Halstead's software science metrics [Halstead 1977]. Whereas cyclomatic complexity is control oriented, the Halstead metrics are text oriented. As well as variations on each of these measures, there are many other program-level measures. In contrast, relatively few measures for assessing design-level complexity have been proposed. Perhaps the most common design-level measures are those developed by Mohanty that are based on a call graph [Mohanty 1976], and basic subtrees, a variation on cyclomatic complexity. Measures for assessing requirements complexity are similarly scarce and not supported by any of the examined tools. Table 7-2 identifies the different types of complexity measures that are provided.

Table 7-2. Supported Complexity Measures

TOOL NAME	Unit Level													Integ Level	
	Basic Blocks (#)	Basic Block (Avg. Len.)	Cyclomatic Complexity	Essential Complexity	Intervals (Ord. 1)	Intervals (Max. Ord.)	Control Flow Knots	Essential Knots	Paths (#)	Paths (Avg. Length)	Paths (Max/Min)	Iteration Groups (Max.)	Halstead's Metrics	Basic Subtrees (S1)	Mohanty's Metrics
ADADL Processor			√												
LDRA Testbed	√	√	√	√	√	√	√	√	√						
Logiscope			√	√									√		√
MALPAS			√												
TCAT-PATH	√		√	√					√	√	√	√			
TestGen			√											√	

Twenty years of theoretical and empirical evaluations have failed to produce consistent, hard evidence of the accuracy of particular measures or on the respective value of alternative measures. Consequently, these measures should be used as indicators, rather than absolute measures of software properties.

7.2 Data Flow Analysis

Data flow analysis is based on consideration of the sequences of events that occur along the various paths through a program. It is used to detect data flow anomalies, of which three types are commonly recognized: (1) a variable whose value is undefined is referenced, (2) a defined variable is redefined before it is referenced, or (3) a defined variable is undefined before it is referenced. While the first of these indicates an actual program defect, the latter two types of anomaly may indicate questionable variable usage rather than specific defects. Since the analysis technique assumes that all paths through the program are feasible, some reported anomalies may be superfluous. Data flow analysis also can be used to categorize procedure parameters as referenced only, defined only, both defined and referenced, or not used.

LDRA Testbed, MALPAS, and EDSA support static data flow analysis. LDRA Testbed performs weak data flow analysis to identify data flow anomalies of the types mentioned above. It also analyzes procedures calls across procedure boundaries to report on procedure parameter usage. MALPAS refines the classification of data flow anomalies. For example, a data variable that is redefined before it is referenced may be classified as either an instance where data is written twice without an intervening read, or as data being written with no subsequent access on a given path. Given a list of procedure input and output parameters, MALPAS compares these with the classes of data to produce a table of possible errors. EDSA uses interactive data flow analysis to facilitate program browsing.

7.3 Control Flow Analysis

Control flow analysis is a process of examining a program structure and identifying major features such as entry and exit points, loops, unreachable code, and paths through a program. This information can be used to determine program complexity and to aid in planning a dynamic test strategy. It can help to decide on strategies for further analysis, for example, to identify where it might be beneficial to partition the code to reduce the number of paths

and, hence, facilitate semantic analysis. The results of control flow analysis can also be used to prepare a diagrammatic representation of the program structure that can aid a user in documenting and understanding a piece of software.

Control flow analysis is provided by the majority of tools that support static analysis. MALPAS, TestGen, LDRA Testbed, and TCAT family all report on unreachable paths. These may be generated as a result of program syntax, for example, as a result of *end if* statements, or the position of a *return* statement. Even though they do not necessarily imply an defect, the occurrence of unreachable paths should be checked. Some of the examined tools go farther. LDRA Testbed, for example, also reports on unreachable branches and other structural units.

Several of the tools use control flow analysis to generate a graphical representation of a program's structure as a logical flow chart or directed graph. This allows visual inspection of program structure and complexity, and can facilitate program understanding at the unit level. AutoFlow-Ada, LDRA Testbed, Logiscope, TCAT, and TCAT-PATH all generate fairly sophisticated graphical representations of a program's structure. AutoFlow-Ada, in particular, provides a user with considerable flexibility in generating a high-quality graphical flow chart. TestGen uses textual facilities to produce a more primitive graph representation. Although MALPAS does not directly produce a directed graph, its list of nodes, with identification of successor and predecessor nodes, helps a user to draw this graph. Graphical representation of the calling relationship between program units also facilitates program understanding. GrafBrowse, LDRA Testbed, Logiscope, and S-TCAT generate call graphs or call trees.

The identification of paths through a program is useful for estimating the resources needed for dynamic analysis and then guiding this testing. AdaQuest, LDRA Testbed, Logiscope, MALPAS, TCAT-PATH, TST, and TestGen all provide this capability. Even more useful, LDRA Testbed, Logiscope, and TestGen explicitly identify the values of logical conditions necessary to cause particular paths to be followed. Logiscope, TCAT, TCAT-PATH, and S-TCAT report on various code statistics. These statistics range from measures such as the number of each type of operator and operand occurring in the software, to measures of the average, minimum, and maximum path length. EDSA provides interactive control flow analysis to facilitate browsing along program paths.

MALPAS, LDRA Testbed, and Logiscope perform structure analysis to verify a program's conformance to the principles of structured programming. Here LDRA Testbed matches templates of acceptable structures with the directed graph of a program on a mod-

ule by module basis. Matching structures are successively collapsed to a single node until either a single node is left, indicating a structured program, or an irreducible state, indicating an unstructured program. MALPAS and Logiscope perform a similar reduction to evaluate the structure.

7.4 Information Flow Analysis

Information flow analysis is used to examine program variable interdependencies. This helps to isolate inadvertent or unwanted dependencies, to indicate how a program can be broken down into subprograms, and to identify the scope of program changes. For security applications, it can be used to aid the identification of spurious or unknown code. Additionally, it supports dynamic testing by identifying which inputs need to be exercised to affect which outputs.

Both LDRA Testbed and MALPAS provide this capability. Currently LDRA Testbed is limited to identifying backward dependencies on a procedure by procedure basis and characterizes variables as strongly or weakly dependent. Future versions of LDRA Testbed will include forward dependencies to identify variables that can be affected by a particular input variable. It will also support information flow dependence assertions to allow comparison of expected dependencies with actual dependencies.

MALPAS identifies all of a program's inputs and examines each executable path to identify dependencies for each output variable. These dependencies include the input variables, constants, and conditional statements on which it depends. It reports on program unit inputs and outputs, which may be more than those passed as parameters. MALPAS also identifies redundant statements.

7.5 Standards Conformance Analysis

Auditors are used to check the conformance of a program to a set of standards. For SDI software, the SPC *Ada Quality and Style: Guidelines for Professional Programmers* [SPC 1991] defines the required standards. Although none of the tools reported here supports these guidelines, ADAMAT discussed in the CRWG study does. Two new tools, Ada-ASSURED and the Ada Quality Toolset, are advertised as providing this support.

LDRA Testbed checks conformance to a set of standards that are of interest to the programming community; this includes much of the Safe Ada Subset. Individual standards can be disabled and the user can weight particular standards or specify acceptance limits, where appropriate. TST reports on conformance to a set of portability standards.

7.6 Quality Analysis

As already mentioned, several tools report on particular quality characteristics such as complexity and compliance with standards. There are, however, many other quality characteristics that provide insight into, for example, code maintainability and portability.

One of the examined tools, Logiscope, employs the Rome Air Development Center (RADC) quality metrics model to allow user-defined quality measurement at three levels of abstraction [RADC 1983]. At the lowest level of the model, the user can defined upper and lower bounds for a predefined set of primitive metrics. Logiscope distinguishes between unit-level metrics and architectural metrics, reporting on both. The user can then specify algorithms to weight and combine the primitive metrics into composite metrics. These composite metrics are, in turn, used to define quality criteria that allow classifying components as, for example, accepted or rejected, based on their computed quality values.

QualGen analyses both design and code complexity and currently interfaces with Lotus 1-2-3 for quality reporting. It provides some 200 primitive metrics which, via Lotus, can be combined into user-defined higher level measures. Software Systems Design, the developer of QualGen, is currently mapping the correspondence of QualGen metrics to the SPC Ada style guide.

7.7 Cross-Reference Analysis

The information acquired from cross-referencing program entities serves many purposes. Perhaps one of the most important of these is identifying the scope of a program change or aiding in the diagnosis of a software failure.

The ADADL Processor provides extensive cross-referencing capabilities. It reports on the cross-referencing between program units, objects, and types. It also reports on the occurrence of *with* and *pragma* statements; the occurrence of interrupts, exceptions, and generic instantiations; and the usage of program unit renaming. LDRA Testbed cross-

references all data items and classifies them as global, local, or parameter and also cross-references procedure usage. Through its browsing capabilities, EDSA provides interactive cross-referencing of data items and Ada objects.

7.8 Browsing

A browser facilitates program understanding by allowing the user to create and present different views of the software. This may include views that show the same piece of software at different stages of development and views that omit some information in order to focus on other details. A browser also may provide the user with the ability to follow the control flow or data flow in browsing through code. These capabilities may be used for several purposes, for example, to aid in reviewing a program or in diagnosing the cause of a software failure.

EDSA focuses on browsing source code at the unit level; it allows browsing forward or backward via data flow or control flow. The user can construct views that suppress or omit irrelevant code details to help him to focus on the concern at hand. Special annotations are available to keep track of the progress of formal code verification. GrafBrowse chiefly operates at the integration level. Here the user can move through graphical invocation hierarchies (or declaration or call-by hierarchies), pulling up the relevant pieces of code as required. The TCAT family of coverage analyzers also allows moving between graphical depictions of program and module structure and the associated source code.

Although not examined in the course of this work, the new version of Logiscope also supports source code browsing.

7.9 Symbolic Evaluation

This type of static semantic analysis provides a more complete examination of a program's operation. Instead of actual input data, symbols such as variable names are used to simulate program execution. This allows the reporting of the mathematical relationships between inputs and outputs for each semantically possible path. It has three primary uses. The relationships can be compared against a program specification to check for consistency. The identified path condition, together with the expression detailing the set or range of input data which causes this path to be executed, supports test data generation. Finally, the

relationships can aid in determination of the expected output for a set of test data. Only MALPAS provides this very useful capability.

7.10 Specification Compliance Analysis

Specification compliance analysis takes semantic analysis a step further by automatically comparing a program against its formal specification to identify deviations. This type of analysis is very powerful, but requires additional work on the behalf of the user.

Here again, MALPAS was the only examined tool that provides this capability. It requires program specification details to be embedded in its intermediate language. (These details may already be available if a formal specification language such as Z, VDM, or OBJ is being used in the development effort.) The output of the compliance analyzer is a set of threat statements that, if the program does not meet the specification, presents the relationships between inputs that cause a deviation to occur.

7.11 Pretty Printing

A useful documentation capability, pretty printing is provided by the ADADL Processor, AutoFlow-Ada, EDSA, LDRA Testbed, and TST.

8. DYNAMIC ANALYSIS

This section reports on the capabilities provided by the examined tools for dynamic analysis where software is evaluated based on its behavior during execution. Dynamic analysis is the primary method for validating and verifying software. Additionally, it is the source of much of the information used in monitoring testing progress and software quality. Traditionally an unstructured and labor-intensive activity, dynamic analysis is a significant cost driver. This study examined the dynamic analysis capabilities of fourteen tools. Table 8-1 identifies the particular functionality provided by each.

Table 8-1. Dynamic Analysis Capabilities of Examined Tools

TOOL NAME	Assertion Analysis	Coverage Analysis				Prof-iling		Timing Analysis	Task Analysis	Test Bed Generation	Test Data Generation				Test Data Set Analysis	Dynamic Flow Graph	Dynamic Call Graph
		Structural (Unit)	Structural (Int.)	Data Flow	Functional	Execution Counts	Instruction Tracing				Structural	Functional	Parameter	Grammar-based			
AdaQuest	F	√				√		√	F								
LDRA Testbed	√	√		F		√	√				√				√	√	√
Logiscope		√	√			√					√					√	√
SoftTest					√							√					
S-TCAT			√			√											
T					√							√					
TBGEN										√							
TCAT		√				√											
TCAT-PATH		√				√					√						
TCMON		√				√		√									
TST						√	√			√			√				
TDGen														√			
TSCOPE[1]		√	√													√	√
TestGen		√				√					√						

1. Used in conjunction with TCAT, TCAT-PATH, or S-TCAT to animate coverage results.

F - Future capability

8.1 Assertion Analysis

An assertion is a logical expression specifying a program state that must exist, or a set of conditions that program variables must satisfy, at a particular point during program execution. Assertion analysis is used to determine whether program execution is proceeding

as intended. In some cases, it may be desirable to leave assertions permanently in the code to provide a self-checking capability. When present in code, even if commented out, assertions can provide valuable documentation of intent.

Of the examined tools, only LDRA Testbed currently supports dynamic assertion analysis. Assertions are embedded in Ada comments and can be used to (1) specify pre- and post-conditions for a section of code, (2) check whether inputs satisfy pre-determined ranges, and (3) check whether loop and array indices are within bounds. Should any assertion fail, a user-tailorable failure handling routine is executed. Assertion checking can be switched on or off, allowing assertions to remain permanently in the code.

8.2 Coverage Analysis

Coverage analysis is the process of determining whether particular parts of a program have been exercised. Its importance is illustrated by academic studies and the experience of the software testing industry that have shown that the average testing group that does *not* use a coverage analyzer exercises only 50% of the logical program structure. As much as half the code is untested and therefore many errors may go undetected at the time of release. By identifying those parts of a program that have not yet been executed, a coverage analyzer can help to ensure that all code is exercised, thus increasing confidence in correct software operation. By measuring the coverage achieved during execution with particular set(s) of test data, these tools also provide a quantitative measure of test completeness. Some tools also aid in determining the test data needed to increase the coverage. Although coverage analyzers do not directly measure software correctness, they are valuable tools for guiding the testing process and monitoring its progress.

There are two basic types of coverage analyzers. Intrusive analyzers instrument code with special statements, called probes, that record the execution of a particular structural program element. The addition of extra code in the program incurs both a size and timing overhead. The alternative, non-intrusive analyzers, requires special hardware and is not addressed in this report.

8.2.1 Structural Coverage Analysis

Several levels of structural test coverage have been proposed. The basic levels for unit testing are statement, branch, and path coverage which require, respectively, each state-

ment, branch, or path to be executed at least once. They impose increasingly stringent levels of testing with statement coverage being the weakest and path coverage the strongest. Since path coverage can be difficult to achieve, various additional levels that lie between branch and path coverage have been proposed. The best known of these additional levels are McCabe's Structured Testing and Linear Code Sequence and Jumps (LCSAJs) [Hennell 1976].

Although unit-level measures can be applied during integration and system testing, they do not provide the additional information that is pertinent at these levels. During integration testing, for example, a measure of the extent to which the relationships between calling and called units has been executed is useful. Functional measures provide a more appropriate measure of test coverage for system testing (see Section 8.2.3).

Table 8-2 summarizes the structural coverage analysis features of the examined tools. As shown in this table, the examined tools vary considerably in the support they provide. The requirements for a test driver to execute the instrumented program is one of these differences. LDRA Testbed and TCMON automatically generate this test driver, as does TestGen under certain circumstances. The generated test drivers also differ. For example, TCMON provides a command-driven test driver that allows the user to explicitly control the handling of generated trace files. Where necessary, both LDRA Testbed and TCMON allow special actions so that this interface can be omitted. There are other significant differences. For example, LDRA Testbed provides different handling of trace data to support host/target testing. It also separates out the data collected from a concurrent program to allow separate reporting for each task.

8.2.2 Data Flow Coverage Analysis

Data flow coverage has been proposed as another measure of test data adequacy. While the traditional structural coverage testing approach is based on the concept that all of the code must be executed to have confidence in its correct operation, data flow testing is based on the concept that all of the program variables must be exercised.

While there are several tools that provide this capability for C programs, production quality tools for data flow testing of Ada code are not yet available. The data flow testing capability of LDRA Testbed, however, is currently under beta testing.

Table 8-2. Structural Coverage Analysis Characteristics

TOOL NAME	Unit-Level Coverage							Integration Coverage	Reporting					Inst. Files Differently	Requires Driver	Limit Instrumentation	Host/Target Support	User-control of Trace Files
	Statement Coverage	Branch Coverage	LCSAJ Coverage	Condition Coverage	McCabe Coverage	Path Coverage	User-specified Coverage		Accumulate Coverage	Animation of Coverage	Frequency Distribution	Warning Level	Identify (Not) Hit Items					
AdaQuest		√					√		√		√		√			√		√
LDRA Testbed	√	√	√	√					√	√	√	√	√	√	√		√	
Logiscope	√	√	√					√	√		√		√	√			√	
S-TCAT								√	√		√		√	√		√		√
TCAT		√		√					√		√		√	√		√		√
TCAT-PATH						√			√				√	√		√		
TCMON	√			√					√		√	√	√	√	√			√
TSCOPE[1]		√				√		√		√	√							
TestGen	√	√			√	√			√				√		√			√

1. Used in conjunction with TCAT, TCAT-PATH, or S-TCAT to animate coverage results.

8.2.3 Functional Coverage Analysis

Functional coverage, which may also be called requirements coverage, provides a measure of the extent to which tests have caused execution of the functions that the software is required to perform. Unlike structural tests, functional tests can determine problems such as the absence of needed code.

Two of the examined tools assess the functional coverage of tests. SoftTest provides a measure of test adequacy in terms of the number of tested functional variations with respect to the number of those testable. T provides a measure of test adequacy based on requirements coverage using user specified pass/fail results. An additional test comprehensiveness measure considers requirements coverage, input domain coverage, output range coverage and, optionally, structural coverage, where each factor can be user-weighted.

8.3 Profiling

Profiling provides a trace of the flow of control during software execution. This information can aid in locating the cause of a failure and the position of the associated defect. Of the examined tools, both LDRA Testbed and TST provides this capability as an optional feature. In the case of LDRA Testbed, however, the Testbed may override the user request if the resulting display exceeds a preset limit.

In general, the majority of computing time is incurred by only a few program segments. This may be because these segments are called frequently, are computationally intensive, or both. When a program needs to be optimized, therefore, it is more efficient to start by identifying where the majority of computing time is spent so that the optimization effort can be appropriately focused. Information on the number of times particular program segments are executed can aid this determination. The coverage analysis tools all give the number of times examined program elements are executed, some additionally identify the number of times each program unit is invoked.

8.4 Timing Analysis

Timing analysis serves several purposes. These range from supporting the validation of software requirements that impose specific timing constraints on software functions to identifying particular program units that consume a significant proportion of computing time.

AdaQuest and TCMON provide timing analysis. Both offer the flexibility of user-specified placement of timers, and measurement using either clock or wall time. TCMON additionally allows a user to request automatic timer instrumentation at the program unit level. This tool reports on the placement of timers (and any counters used for structural coverage analysis) to provide information that can be used to estimate the influence of instrumentation statements on measured time.

8.5 Test Bed Generation

Unit and integration testing require the ability to invoke the appropriate modules, passing necessary inputs and capturing the actual outputs so that they can be compared against expected outputs. Integration testing may proceed in either a top-down or bottom-up man-

ner. In the first case, testing starts with the most abstract, or high-level modules and requires the use of *stubs* to represent those modules called by the module under test. In bottom-up testing, the most detailed, or lower-level, modules are tested first. Here test *drivers* are required to simulate the modules that invoke the modules under test. Development of such test drivers and stubs can be complex and greatly facilitated by automated support. In addition to eliminating the need for much manual labor, automatic generation also promotes a standardized testing environment.

LDRA Testbed, TCMON, and TestGen all generate the test drivers needed for execution of an instrumented program. These are, however, very limited drivers primarily intended to handle the trace files used to collect coverage details. Of the examined tools, TBGEN and TST are the only ones that provide true test bed generation, and only TBGEN supports stub generation. Table 8-3 summarizes the test bed generation characteristics of these two tools.

Table 8-3. Test Bed Generation Characteristics

TOOL NAME	Driver Generation	Stub Generation	Restrict Included Entities	Support Unknown Entities	Command Language					Record Keeping				
					Output Assertions	Test Bed Variables	Control Constructs	Expressions	Breaks	Execution Log	I/O Tracing	Execution Statistics	Scripting	User Input Recording
TBGEN	√	√	√	√	√	√	√	√	√	√	√	√	√	√
TST	√				√									√

8.6 Test Data Generation Support

Dynamic analysis requires software to be executed with a set of test data. The resulting outputs are then captured and compared with the outputs expected for the given input data. The traditionally manual and labor-intensive method of preparing test data has typically limited the extent of testing that is performed. Although the available tools do not totally replace the human effort required, they can make a substantial reduction to the amount of human labor needed.

As mentioned above, dynamic analysis requires comparing expected results against actual results to determine the success or failure of a test. Determining expected results is another traditionally manual and difficult task. Research into tools, called *oracles*, to automate this task has been ongoing for many years. As yet, however, symbolic evaluators (see Section 7.9) come the closest to supporting this capability.

8.6.1 Structural Test Data Generation

During testing, there are occasions where it is necessary to determine the test data that will cause a specific branch or path to be executed. This occurs, for example, when it is necessary to attain a specified level of structural coverage and existing test data has not executed some structural elements.

Support for this activity is available at two levels. AdaQuest and TCAT explicitly identify the program segments that comprise particular program branches and paths. LDRA Testbed, Logiscope, TCAT-PATH, and TestGen provide the same information and, additionally, explicitly identify the conditions required to cause each structural element to be executed.

8.6.2 Functional Test Data Generation

Functional tests can be derived from a requirements specifications using three categories of methods: (1) algorithmic techniques such as cause-effect graphing, equivalence class partitioning, and boundary value analysis; (2) heuristic techniques including fault directed testing and the traditional error guessing; and (3) random techniques that employ random generation of test data.

T supports all these techniques. Additionally, it is capable of incremental test data generation, that is, tests can be generated for software changes only. T is the only examined tool that produces test data values ready for immediate use in testing.

SoftTest supports cause-effect graphing to compile a database of input conditions for each unique function. The user then works from these conditions to determine the necessary test data. In those cases where identified functions are not directly testable, for example, because results produced by one function may be obscured by other functions, SoftTest identifies intermediate results that, if observable, would enable otherwise obscured functions to be tested.

8.6.3 Parameter Test Data Generation

Thorough test coverage at the integration level requires that each subprogram be executed over a range of parameter values. Of the examined tools, only TST provides automated generation of test data for certain types of subprogram parameters. This generation occurs in one of two forms. The user can specify that all possible values for a parameter be generated (or first and last values for floating point numbers). Alternatively, the user can request that these values are divided into a number of partitions and that the first, middle, and last values from each partition be selected.

8.6.4 Grammar-based Test Data Generation

In those cases when the test data is simply structured, and this structure is amenable to description, grammar-based test data generation allows rapid, automated generation of large amounts of test data. This capability is particularly useful in random testing.

TDGen provides this functionality. Test data is generated according to location-specific data, uniformly distributed data, or value-factored data. TDGen can generate data randomly, sequentially, or according to a user specification.

8.7 Test Data Analysis

Two types of test data analysis are considered here. In the first case, test data sets are analyzed to identify which test data sets execute which lines of code. When particular lines of code are changed, this information shows which test data sets are affected by the change and must be rerun. The second type of test data analysis detects and reports on redundant test data sets. This identifies test data sets that are essentially equivalent in effect and, therefore, can be eliminated to reduce testing cost without affecting test effectiveness.

LDRA Testbed is the only identified tool that supports these capabilities. The analyses are performed on data collected during structural coverage analysis.

8.8 Dynamic Graph Generation

A visual representation of the execution flow of a program can aid in understanding that program and diagnosing the cause of failures. LDRA Testbed and Logiscope provide this

facility at both the unit and integration levels. TSCOPE uses the outputs of TCAT or TCAT-PATH to animate the execution coverage on a directed graph; and the output of S-TCAT can be used to animate coverage on a call tree representation of the program under test.

9. FINDINGS

This study examined a number of software testing tools to the extent necessary to gain a feel for their capabilities. However, none of the tools was examined in great depth. Only tools supporting test management, problem reporting, and static and dynamic analysis of Ada code were considered. Categories of tools such as regression analyzers and emulators were ignored. Additionally, some promising tools that may fill some of the identified functional gaps are still awaiting examination.

9.1 Status of Available Tools

Reviews of testing practices and tool usage reveal extremely poor exploitation of available testing tool support. In the last ten years software developers have placed much focus on software development tools and there has been an explosion in the availability of CASE systems and other types of development environments. Only in the last few years, however, has much attention been paid to testing tools. These tools are now starting to come to market in increasing numbers. Even so, available evidence suggests that they are seldom used.

As the number of available testing tools has increased, some trends are emerging. Most noticeably, there is an increased focus on test management and a movement towards customer-oriented measures of software quality. On the technical side, there is a movement towards graphical user interfaces using windows. There is, however, no evidence of increased standardization in terms of testing functionality. Even within one category, no single tool provides all desirable functionality, and different tools support different groups of functions. These functional differences require a potential user to perform tool comparisons with caution and to select a tool very carefully.

The following findings relate to the potential for the use of testing tools in the development and support of SDI software.

- **Test management.** Test management tools offer critically needed support for test planning and test progress monitoring. This category of test tool is perhaps the latest to come to market. The capabilities provided for capturing test plans, test procedures, and test cases are generally similar. Capabilities for capturing software requirements, tracing these to particular tests, and supporting change impact analysis, however, vary significantly. With the exception of reliability analysis tools, which are becoming more common, progress monitoring is seldom available and primitive. Similarly, only one tool that supports cost reporting was identified and the analysis performed is also

primitive. Nevertheless, the ability of these tools to manage a collection of test information is very valuable and, even though its analysis could be improved, the data available from this analysis is urgently needed to support the management and documentation of test activities.

- **Problem reporting.** In addition to their primary use in tracking identified software problems and managing problem resolution, problem reporting tools offer support for test management. They provide information on the status and quality of software products; in particular, they capture the data needed for software reliability modeling. This data can also provide valuable insights into the status and quality of the software development processes themselves, and so support continuous process improvement.

  Problem reporting tools fall into two classes. The network-based class of tools are intended for use on multiple, geographically dispersed projects. They offer specific support for customer submission of problem reports and provide automatic notification of changes in problem status. Those tools that provided problem reporting as one part of test management capabilities run on a stand-alone personal computer but capture much of the same types of problem information and provide similar analyses of problem data. There are several problem reporting tools that could be brought into immediate use, although some thought should be given to defining a standardized set of problem data to be captured across all SDI software development efforts.

- **Static analysis.** Available static analysis tools are essentially limited to facilitating program understanding and assessing characteristics of software quality. They provide some minimal support for guiding dynamic testing. Static analysis requires little in the way of test environment set up and a minimum of human intervention. It can detect the presence or absence of certain, limited types of defects and allows these defects to be detected reliably and early in the testing process. The types of defects traditionally found by static analysis tools, however, are now routinely checked for by Ada compilers. Currently, one of the main values of static analysis tools is in supporting an understanding of software and guiding dynamic testing. Quality analysis is a particular type of static analysis where assessment of a set of predefined quality characteristics can be used to provide early indication of general software quality and the identification of potential problem areas.[3]

  In the types of tools examined, complexity analysis and control flow analysis are the most common static analysis functions supported. A couple of examples of data flow analysis tools have appeared and are expected to become more common in the future. Two types of tools to aid a user in understanding and documenting a piece of code are available: graph generators and browsers. Flow graph and call graph generations are

3. The role of quality analysis is discussed extensively in the GPALS Software Quality Program Plan [GPALS 1992a].

quite common, although they vary greatly in the quality of the representations used to present these graphs. A few browsers are currently available and these are expected to become more common over the next few years. This study and the CRWG's study of quality analysis tools have, between them, identified several tools that check conformance of code with a set of project standards. One of these, ADAMAT, checks for conformance with the SPC Ada style guidelines. Two new tools that also support the SPC standards have recently been identified. More advanced types of static analysis, such as symbolic evaluation, are uncommon.

- **Dynamic analysis.** Although many needed dynamic analysis capabilities are infrequently available, tools are available that offer considerable support for dynamic testing to increase confidence in correct software operation. Dynamic analysis is the principle method used for software validation and verification. Here automated support for the preparation of a test bed, generation of test data, and analysis of test results is needed. Tools that provide this functionality will decrease the cost of testing by increasing the productivity of the human tester and increase software quality by supporting such activities as test data adequacy assessment.

  Structural coverage analyzers and profilers are the most common dynamic analysis tools and are widely available on a range of operating platforms. The structural coverage analyzers generally focus on statement and branch coverage, that is, relatively low coverage measures. Support for path coverage analysis and structural coverage at the integration level is less frequently available.

  Support for other types of dynamic analysis is also infrequently available. Only two of the examined tools provide timing analysis. Only two tools offer test driver generation for bottom-up testing, and only one of these also generates the stubs needed for top-down testing. Few tools support test data generation for structural or random testing, although two tools that support the generation of functional test data from a requirements specification have been introduced. Assertion testing is a relatively new capability that is, as yet, only provided by one tool.

Tools of similar types vary widely in the capabilities they provide and in characteristics such as tailorability and robustness. In general, the examined tools require little sophistication on the part of the user and are supported by good documentation. Some actively guide a user through necessary tasks, keep a record of test activities, and take extra steps to relieve the user of repetitive tasks. In general, however, the tools employ primitive user interfaces that could benefit from the application of human factors engineering. In several cases, the need to refer to separate listings to identify objects referenced in reports complicated tool use. There were instances where different tools gave different results when performing the same function, for example, calculating cyclomatic complexity. Moreover, some of the

tools contained faults. While most failures were trivial, others rendered a tool unusable until fixed by the supplier. In three cases, major failures occurred when using the tool on sample software supplied by the supplier. Consequently, prospective tool users should carefully consider a tool's usage history and the types of support options provided by the tool supplier.

9.2 Significant Deficiencies

Available testing tools offer significant opportunities for increasing software quality and reducing development and support costs. Even so, there are a number of problems with these tools and a lack of needed functionality that may handicap SDI software testing. The following problems are of particular concern.

- **There is a lack of support for testing concurrent Ada software.** The vast majority of current testing techniques are intended for testing sequential code. Concurrent software, however, introduces special concerns. The inherent indeterminism of concurrent programs means that two executions of the same software with the same inputs can produce different behaviors. This lack of reproducibility handicaps, for example, determining the cause of a failure and retesting a modified program. Concurrent programs can also contain a new class of faults, called synchronization faults. Additional tests are needed to check for existence of these faults. Testing techniques addressing these issues are appearing, along with some prototype tools, such as AdaTDC being sponsored by the National Science Foundation. One commercial tool that is expected to support concurrent re-execution within the next year is PARTAMOS, under development by Alcatel Austria. Meanwhile, the majority of the commercial Ada-based static and dynamic analyzers are capable of recognizing all the concurrent Ada language features, but not fully acting on them.
- **There is a need for increased tool integration to provide more complete coverage of testing activities.** The majority of tools provide support for a specific, limited set of testing activities. No single tool, or supplier toolset, provides all desirable functionality. While tools that support different types of activities can generally be used together, simply applying them independently in sequence is usually not the most cost-effective approach. It can incur unnecessary duplication of both human and computer work and may require additional steps to make the output of one tool acceptable to another. It also requires users to gain familiarity with a number of different user interfaces, as well as requiring system administrators to support a number of independent tools. Moreover, true *functional integration* requires some common, underlying model of the software development process model. For example, a test log automati-

cally captured by a test bed during test execution activities should be the same log that a test management tools uses in monitoring test progress. This type of integration would greatly increase the power of available tools and their ease of use.

- **There is a need for integration of testing tools into CASE systems to provide improved feedback into development activities.** While some CASE systems do provide support for code level testing, this support is generally less extensive than that provided by stand-alone testing tools. At the same time, CASE tools are providing more support for testing activities during early development phases than stand-alone tools. A more careful look at the testing capabilities of current CASE systems is needed, together with an evaluation as to which, and how, independent tools should be integrated with them to provide a comprehensive test environment. Here again, functional integration into a CASE requires that test processes and products are themselves integrated into the underlying software development process model. These issues have yet to be addressed.

There is a lack of data on the cost effectiveness of particular test techniques and tools that can be used to encourage and guide their use. Although there have been many studies into the comparative value of certain test techniques, there is a lack of data on the practical costs and benefits of particular testing techniques, and the tools that support those techniques. This information is needed to determine, for a given set of circumstances, the most appropriate techniques and tools to apply, the order in which to apply them, and the extent of that use. It is also needed during planning activities, to support the estimation of needed testing resources, and in monitoring test progress. The data captured in test logs and problem reports can be used for this purpose, imposing a minimal data collection burden on software developers. Where this data is maintained automatically, it will be a simple task to forward it to a central site for analysis, such as the Level 2 System Simulator (L2SS) software metrics database.

A number of promising testing techniques have been proposed in the last decade that have failed to progress beyond prototype status. One example of this is the software fault tree analysis used for error cause and effect analysis in support of risk management. The Anna toolset is an example of a suite of prototype tools for assertion-based testing of Ada code. Further development of such techniques and supporting tools could start to fill some of the gaps in needed testing capabilities.

Additionally, there are needed automated test capabilities that are provided for other languages but not available for Ada. Examples of capabilities available for other languages include error seeding as another measure of test data adequacy; support for test coverage

analysis of kernel, daemon, and library code in addition to application code; and critical path analysis. Similarly, while several tools supporting data flow testing of C code are available, only one tool supplier with plans to provide data flow testing of Ada code has been identified. Here again, further tool development could start to fill some of the gaps in needed testing capabilities.

PART II

TOOL EXAMINATION REPORTS

10. INTRODUCTION

This part of the report describes the selected tools in terms of their usage. Tools are grouped by supplier and the report details the operating environment and the functionality provided. Where applicable, price information, accurate at the time of examination, is also included. Each description is supported with observations on ease of use, documentation and user support, and Ada restrictions. Problems encountered during the examinations provided insight into the reliability and robustness of each tool. Each description is accompanied by sample outputs.

Table 10-1 summarizes the details given for each tool. It also identifies available bridges between testing tools and CASE systems. Table 10-2 presents relevant supplier data.

Table 10-1.Tool Profiles

TOOL NAME	TOOL STATISTICS				OPERATING ENVIRONMENT												LINKS TO OTHER TOOLS	
					Machine			O/S										
	First Marketed	Examined Version	Number Users/Sites	Starting Price	PC Machines	Workstations	Other	Unix	VMS	DOS	Windows Supported	Network Version	Graphics Capability	Import Formats	Export Formats	User Interface	Testing Tool Bridges	CASE/Other Bridges
ADADL Proc.	1984	5.3E	>200 S	$5,000	√	√	√	√	√	√	O					M		
AdaQuest	1991	1.1	<5 U	$6,500		√	√		√							C		CASE framework
AutoFlow	1992	1.02	<4,000 U	$9,950	√			F		√	F		F		ASCII, HPGL, PIC, Postscript	C		ADW, IEW
DDTs	1989	2.1.6	>100 S	$6,000		√	√	√			F	√		Converters	Converters	B	via QTET	HP Softbench, RCS, SCCS
EDSA	1991	2.0	<10 U	$3,750	√	√	√	√	√	√	√					M		RCS, SCCS
GrafBrowse	1991	2.2.2	<10 S	$5,500	√	√	√	√	√	√	√		√		Postscript, HPGL	M		
LDRA Testbed	1974	4.8.01	>400 S	$12,000	√	√	√	√	√	√	O	√	√	Defined formats	Defined formats, Postscript	M	TBGEN	Mascot, STP, System Engineer, Teamwork...
Logiscope	1985	3.2	>5,000 U	$14,000		√	√	√	√		√		√	ASCII	ASCII, DecWrite, HPGL, Interleaf, Postscript, pcl, imPRESS, Genicom	B		DecFuse, HP Softbench, Software Back Plane
MALPAS	1986	5.1	>50 U	$60,000			√		√							C		
Metrics Manager	1989	2.02	>30 S	$14,950	√					√	F		√	ASCII		M	Test/Cycle	Project Manager Workbench
QES/Manager	1991	2.2	>50 U	$2,500	√					√	F	√		Converters	Converters	M	QES/Architect	
QualGen	1988	1.1	<10 S	$4,000	√	√	√	√	√	√	O		F		Lotus	M		
SQA:Manager	1990	2.0	>100 U	$995	√	√		√		√	√	√	√	Converters		M	SQA:Robot	
SRE Toolkit	1990	3.11	>200 U	$995[a]	√	√		√		√			√	Defined formats	PIC	C		
SoftTest	1987	3.1	>100 U	$2,500	√					√						M	AutoTester, Gate, Automator qa, SQA:Robot, TestPro, V-Test, X/TestRunner, Microsoft Test for Windows,	

Table 10-1 continued: Tool Profiles

TOOL NAME	TOOL STATISTICS				OPERATING ENVIRONMENT											LINKS TO OTHER TOOLS	
					Machine			O/S									
	First Marketed	Examined Version	Number Users/Sites	Starting Price	PC Machines	Workstations	Other	Unix	VMS	DOS	Windows Supported	Network Version	Graphics Capability	Import Formats	Export Formats	User Interface	Testing Tool Bridges	CASE/Other Bridges
																	Workstation Interactive Test Tool for OS/2	
T	1987	3.0	>300S	$7,000	√	√	√	√	√	√	√	√			ASCII, IEEE 1175	B	AutoTester, Ferret, XRunner	Excelerator, IEF, StP, Teamwork
T-PLAN	1989	2.0	>120U	£9,500	√	√	√	√		√	O	√		DIF, dBASE, Lotus	DIF, dBASE, Lotus	M	Automator qa	
TCAT	1986	1.3	>2,000 U	$4,900[b]	√	√		√		√	O		O			B	TSCOPE	HP Softbench, DEC FUSE
TCAT-PATH	1988	8	>2,500 U	$4,900[b]	√	√		√		√	O		O			B	TSCOPE	HP Softbench, DEC FUSE
S-TCAT	1987	1.3	>2,500 U	$4,900[b]	√	√		√		√	O		√			B	TSCOPE	HP Softbench, DEC FUSE
TDGen	1990	3.2	~500 U	$500	√	√		√		√	O		O			B		
TSCOPE	1989	1.2	>2,500 U	$4,900[b]		√		√			√		√			M	TCAT, S-TCAT, TCAT-PATH	
TBGEN	1986	3.1	>30 S	$2,850	√	√	√	*	√	√	O					C	LDRA, TCMON	DDC-I CASE Toolbox*
TCMON	1986	2.2	>30 S	$2,300	√	√	√	*	√	√	O					C	TBGEN	DDC-I CASE Toolbox*
TST	1989	2.0	>3	Gov.	√		√		√	√						M		
Test/Cycle	1990	3.02	>10 S	$3,500	√					√	√		√		ASCII, Postscript, GKS, Tektronix	M	Metrics Manager	Ad/Cycle, Project Manager Workbench
TestGen	1984	2.2.2	>1,000 U	$4,600	√	√	√	√	√	√	O		O			M		Excelerator, StP, Teamwork

F - Capability under development
U - Users
S - Sites
O - Optional

M - Menu drive
C - Command driven
B - Menu and command driven

* - Version marketed by DDC International
[a] - Price for 3-day training course
[b] - Sold as group

Table 10-2. Supplier Profiles

SUPPLIER NAME	SUPPLIER STATISTICS			SERVICES PROVIDED							Examined Tools	RELATED SUPPLIER TOOLS						
	Phone Number	Supplier Size	Established	Consultancy	Training	Test Planning	Test Performance	User Group	Newsletter	Hot-Line Support		Tool Name	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis	Regression Analysis	CASE
Array Systems Computing, Inc.	(416) 736-0900	<50	1981	√	√		√		√	√	EDSA							
AutoCASE Technology	(408) 446-2273	<10	1988	√						√	AutoFlow-Ada							
Bender & Associates	(415) 924-9196	<10	1977	√	√	√				√	SoftTest							
Computer Power Group, Inc.	(708) 574-3030	>3,000	1968	√	√	√	√	√	√	√	Test/Cycle, Metrics Manager							
General Research Corp.	(805) 964-7724	~1,000	1960s	√	√		√				AdaQuest	J7AVS RXVP80			√ √	√ √		
Program Analysers, Ltd.	+44 0635-528828	31	1987	√	√			√	√	√	LDRA Testbed							
Programming Environments, Inc.	(908) 918-0110	<20	1981		√				√	√	T	Runner				√		
Quality Engineering Software, Inc.	(203) 278-7252	<20	1991	√	√	√				√	QES/Manager	QES/Architect QES/Qease QES/Expert QES/Programmer				F F	√ F	
QualTrak Corp.	(408) 274-8867	12	1986	√	√			√	F	√	DDTs	RDDTs QTET	F	√			F	
TA Consultancy Services, Ltd.	+44 252-711414	120	1974	√	√			√	√	√	MALPAS							
STARS Foundation Repository	(304) 594-9817	9	1989								TST							
Software Quality Assurance, Ltd.	+44 0277-374415	18	1984	√	√	√	√	F	F		T-PLAN							
Software Quality Automation	(800) 228-9922	15	1989	√						√	SQA:Manager	SQA:Robot					√	
Software Quality Engineering	(800) 423-8378	5	1985	√	√	√					SRE Toolkit							
Software Research, Inc.	(415) 957-1441	>25	1977			√		√	√	√	√	TCAT, S-TCAT, TCAT-PATH, TDGen, TSCOPE	SMARTS CAPBAK EXDIFF STATIC TRACKER	√		√		√ √ √	
Software Systems Design	(714) 625-6147	<10	1985	√	√					√	TestGen, QualGen, GrafBrowse, ADADL Processor	AISLE Toolset			√			√
Testwell Oy	+358 31-165464	<5	1992							√	TBGEN, TCMON							
Verilog USA	(214) 241-6595	<20	1986	√	√			F		√	Logiscope	AGE/ASA AGE/GEODE DocBuilder			√ √	√ √		√ √ √

F - Future capability

11. AdaQuest

The AdaQuest toolset provides a variety of static and dynamic techniques for testing Ada software. It is based on two earlier verification and validation systems, RXVP80 and J7AVS, that, respectively, support Fortran and Jovial testing.

The static analysis capabilities of the current version of AdaQuest are limited to identifying program branches and the lexical nesting structure of specified compilation units. Existing dynamic capabilities consist of coverage and timing analysis.

11.1 Tool Overview

AdaQuest was developed by General Research Corporation. The first version of this toolset, version 1.1, became available in December 1991. It runs on VAX/VMS platforms. At the time of evaluation, the price for AdaQuest started at $6,500.

AdaQuest requires that code to be analyzed resides in an AdaQuest program library. Each library is associated with a VMS directory that contains the intermediate files of the relevant compilation units. Several library management commands are provided. These include commands to set a *current* library and build a *working set* of compilation units. Special facilities are provided for reading source files into a library; in the current version of the toolset, source files are limited to containing a single compilation unit.

The AdaQuest Analyzer generates branch reports and unit nesting reports for user-specified library units. In the first case, the result is an annotated source code listing that identifies and numbers each decision branch in each program unit of a specified compilation unit. This report is needed to select locations for the insertion of coverage and timing probes (see below). It is also required for interpretation of branch coverage reports. The unit nesting report shows the lexical nesting of the program units in a compilation unit.

Each program unit can be instrumented to collect either coverage data, timing data, or both. The user specifies the library unit bodies and subunits to be instrumented, and each instrumented unit is written to a separate file. Instrumentation is performed by inserting special statements into the source code. Where necessary, individual source code statements are first transformed so that these insertions will be syntactically legal. An *exit* statement, for example, may be replaced with *if* and *goto* statements.

Two different types of probes are available to collect coverage data. Branch coverage probes are inserted automatically at each branch point (including at the start of each *accept* and *block* statement). Any code transformations necessary to ensure correct coverage measurement are also made automatically. The second type of probe, called test case probes, are used to partition the data collected from an instrumented program. They allow, for example, measuring the coverage achieved in each execution of a loop. Test case probes are inserted at user-specified points in the source code and take the form of procedure call statements. It is the user's responsibility to ensure that these are placed in a syntactically legal fashion; AdaQuest does not check the placement. The resulting instrumented files include file header information that identifies the unit, original source file, and type of instrumentation performed. They are accompanied by files containing two additional AdaQuest-generated units needed for the collection of coverage data.

For timing analysis, probes are also inserted at user-specified locations. In each case, the user gives a *start* and *stop* location in the form of source code line numbers; these locations may reside in different program units within a compilation unit. Again, it is the user's responsibility to ensure that insertions at these defined locations will be syntactically legal. The user also specifies whether data should be measured in terms of CPU or wall clock time.

Compilation and linking of the instrumented program is performed using the standard VAX facilities. AdaQuest does, however, provide a compilation script that can be used to compile the two AdaQuest-generated run-time units. When executed, the instrumented program collects coverage and timing data in an automatically created trace file. (As with the other tools that write coverage details to a trace file, a program run must terminate normally so that the trace file is closed by the operating system.) If desired, the user can allocate a name and description to the trace file.

AdaQuest maintains a test history for each body or subunit in a library in order to allow reporting on the cumulative coverage achieved over a series of test runs. Initially empty, the user specifies when a trace file should be appended to the appropriate histories. Normally, the test history for a unit is cleared when the program library is updated. In certain circumstances, the user can override this function to keep a history, although this ability must be used with great care.

The reports that are available can be produced for all or only user-specified program units. The Test Run Report identifies the original source code file(s) and indicates how it was instrumented. Coverage reports are generated using data from a single trace file, called

the current test run, and, in most cases, user-specified test histories. Between them, the coverage reports provide counts of the number of times each program unit, accept statement, and block statement was executed, counts of the number of times each conditional branch was executed, the execution status (first-time hit, never hit) of each branch, and the percentage coverage of the branches in each unit. In some cases, histograms are provided to compare the execution counts of different items. Two additional reports can be generated using coverage information from the test history files alone. Timing analysis reports are generated from a single trace file. They detail the timing probe placement, the number of times each selected code segment was invoked, and the minimum, maximum, and average time taken for each segment. Timing data is not accumulated in test histories.

11.2 Observations

Ease of use. The user interacts with AdaQuest through a command interface. This interface requires considerable memorization on the user's part (the Analyzer, for example, has some 27 different commands) and exhibits some inconsistencies. Although the ability to explicitly specify the locations for coverage and timing probes can be valuable, the need to manually refer to the annotated source code listing is tedious and a possible source of error. This inconvenience could be reduced by providing, for example, some automatic insertion of timing probes to measure the time spent in named units. Output listings are handled in an unusual manner; all commands that produce an output listing automatically invoke the VAX *edlin* editor.

At the time of evaluation, the on-line help provided summary information on only a small number of available commands, although this should be a useful feature when completed.

Documentation and user support. A complete AdaQuest user manual was not available at the time of examination. The documentation that was provided, however, was well-written and easy to follow. One nice feature is a command dictionary that provides a useful reference manual. Tool installation was straightforward.

Instrumentation overhead. AdaQuest allows the user to control the extent of instrumentation by requiring the user to explicitly identify the units to be instrumented. Two special run-time units are provided that must be included in an instrumented executable to handle the creation and recording of a trace file. Including these special units, full instru-

mentation for branch coverage of the Ada Lexical Analyzer Generator gave an approximately 19% increase in the total source code size.

Ada restrictions. AdaQuest supports the full Ada language, including extensions described in Chapter 13 of the Ada Language Reference Manual. The only exception is the Ada *terminate* alternative (see LRM 9.7.1) which contains no statements and cannot be instrumented.

Problems encountered. No problems were encountered during the examinations of this tool. AdaQuest operated exactly as described in the documentation provided.

11.3 Planned Additions

Future versions of the static analyzer are expected to generate dependency reports and check for logic errors (such as infinite loops, unreachable statements, and uninitialized variables). Conformance checking against standards relating to the use of forbidden constructs and those specifying maximum/minimum constraints on the quantity of Ada constructs appearing within a certain scope is also anticipated. A source code profiler will list any non-zero counts for some 228 Ada features. AdaQuest is also expected to include a query facility that provides direct access to this data for quality analysis tools.

Additional dynamic analysis capabilities expected to become available include the use of assertions for checking unit- and interface-level design constraints. Finally, a task analyzer is also planned that traces the actual synchronization relationships between Ada tasks, creating timing diagrams to help in diagnosing synchronization errors such as deadlock and starvation.

11.4 Sample Outputs

Figures 11-1 through 11-12 provide sample outputs from AdaQuest.

```
27-JAN-1992 10:45        ADAQUEST UNIT NESTING REPORT                PAGE    1
Library: ADALEX;WORK                                  Comp Unit: LL_COMPILE:BODY
                                                                27-JAN-1992 10:35:12
Structure Unit                          Unit Kind                 Starting Line
LL_COMPILE                              Procedure Body                   26
 ..LLNEXTTOKEN                          Procedure Spec                  136
 ..LLFIND                               Function Body                   140
 ..LLPRTSTRING                          Procedure Body                  165
 ..LLPRTTOKEN                           Procedure Body                  177
 ..LLSKIPTOKEN                          Procedure Body                  190
 ..LLSKIPNODE                           Procedure Body                  203
 ..LLSKIPBOTH                           Procedure Body                  217
 ..LLFATAL                              Procedure Body                  234
 ..GET_CHARACTER                        Procedure Body                  246
 ..MAKE_TOKEN                           Function Body                   264
 ....CVT_STRING                         Function Body                   271
 ..LL_TOKENS                            Package Spec                    326
 ....ADVANCE                            Procedure Spec                  328
 ..LL_TOKENS                            Package Stub                    334
 ..LLNEXTTOKEN                          Procedure Body                  337
 ..LLTAKEACTION                         Procedure Stub                  349
 ..LLMAIN                               Procedure Body                  352
 ....READGRAM                           Procedure Body                  383
 ......BUILDRIGHT                       Procedure Body                  389
 ......BUILDSELECT                      Procedure Body                  449
 ....PARSE                              Procedure Body                  499
 ......ERASE                            Procedure Body                  505
 ......TESTSYNCH                        Procedure Spec                  522
 ......EXPAND                           Procedure Body                  525
 ........MATCH                          Function Body                   533
 ......TESTSYNCH                        Procedure Body                  593
 ........SYNCHRONIZE                    Procedure Body                  596

27-JAN-1992 10:45        ADAQUEST UNIT NESTING REPORT                PAGE    2
Library: ADALEX;WORK                          Comp Unit: LL_COMPILE.LL_TOKENS
                                                                27-JAN-1992 10:39:35
Structure Unit                          Unit Kind                 Starting Line
LL_TOKENS                               Package Body                     25
 ..ADVANCE                              Procedure Body                   49
 ....GET_CHAR                           Procedure Body                   56
 ....CHAR_ADVANCE                       Procedure Body                   69
 ....LOOK_AHEAD                         Procedure Body                   86
                                  ...
```

Figure 11-1. AdaQuest Unit Nesting Report

```
27-JAN-1992 10:45          ADAQUEST BRANCH REPORT                    PAGE    1
Library: ADALEX;WORK                                  Comp Unit: LL_COMPILE:BODY
                                                      27-JAN-1992 10:35:12

                                    ...
  140 function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
  141     --  Find item in symbol table -- return index or 0 if not found.
  142     --  Assumes symbol table is sorted in ascending order.
  143     LOW, MIDPOINT, HIGH: INTEGER;
  144 begin
***** BRANCH 1  PROGRAM UNIT START
  145     LOW := 1;
  146     HIGH := LLTABLESIZE + 1;
  147     while  LOW /= HIGH  loop
***** BRANCH 2  LOOP TEST FAIL
***** BRANCH 3  LOOP TEST PASS
  148       MIDPOINT := (HIGH + LOW) / 2;
  149       if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
***** BRANCH 4  IF TEST PASS
  150         HIGH := MIDPOINT;
  151       elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
***** BRANCH 5  ELSIF TEST PASS
  152         if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
***** BRANCH 6  IF TEST PASS
                                    ...
  726 begin  -- LL_COMPILE
***** BRANCH 142  PROGRAM UNIT START
  727     CREATE (STANDARD_ERROR, OUT_FILE, "std_error", "");
  728     LLMAIN;
  730     CLOSE (STANDARD_ERROR);
  731 end  LL_COMPILE;
-------------------------------------------------------------------------------

                                                    BRANCH SUMMARY
Branch Kind                      Branch ID    Begin     End       Branch Statement Path
-------------------------------------------------------------------------------
PROGRAM UNIT START                     1      145       147       145    146    147
LOOP TEST FAIL                         2      147       161       147    161
LOOP TEST PASS                         3      147       149       147    148    149
IF TEST PASS                           4      149       147       149    150    147
ELSIF TEST PASS                        5      149       152       149    152
IF TEST PASS                           6      152       153       152    153
                                    ...
PROGRAM UNIT START                   142      727       730       727    728    730
                                          * = Branch contains an Infinite Loop
Unreachable Statements => -- NONE --
-------------------------------------------------------------------------------
                                    ...
```

Figure 11-2. AdaQuest Branch Report

```
=============================================================================
27-JAN-1992 12:13          ADAQUEST TEST RUN REPORT                  PAGE    1

=============================================================================
-----------------------------------------------------------------------------
Trace File       : USR:[ADATEST]ADAQUEST.ETF;16
Time of Run      : 27-JAN-1992 12:08:39
Name             : EXAMPLE_1
Description      : Run with 1st test file
# of Test Cases : 1
-----------------------------------------------------------------------------

Test Run Units :

LL_COMPILE:BODY
   Instrumented From File :
      USR:[ADATEST.ADALEX2]LL_COMPILE.ADA;10
   Instrumentation Parameters :
      Coverage

LL_SUPPORT:BODY
   Instrumented From File :
      USR:[ADATEST.ADALEX2]LL_SUP_BODY.ADA;1
   Instrumentation Parameters :
      Coverage
```

Figure 11-3. AdaQuest Coverage Test Run Report

```
27-JAN-1992 12:14                    ADAQUEST UNIT COVERAGE REPORT                    PAGE    1
Library: ADALEX;WORK

                                  - - - TEST RUN INFORMATION - - -

Trace File    : USR:[ADATRST]ADAQUEST.ETF;16
Time of Run   : 27-JAN-1992 12:08:39
Name          : EXAMPLE_1
Description   : Run with 1st test file
# Test Cases  : 1

                                  - - - REPORT QUALIFIERS - - -

By Test Case          : No
```

```
27-JAN-1992 12:14                    ADAQUEST UNIT COVERAGE REPORT                    PAGE    2
Library: ADALEX;WORK

                                                                  Invocations - Normalized to Maximum
                                                                  0       20        40        60        80       100
Line#   Kind          Program Unit                  Invocations   |---------|---------|---------|---------|---------|

| LL_COMPILE:BODY
| 27-JAN-1992 10:35:12
   26  PROC BDY  LL_COMPILE                                  1   *
  140  FUNC BDY  LLFIND                                    198   **************************
  337  PROC BDY  LLNEXTTOKEN                               134   ******************
  352  PROC BDY  LLMAIN                                      1   *
  383  PROC BDY  READGRAM                                    1   *
  389  PROC BDY  BUILDRIGHT                                 64   ********
  449  PROC BDY  BUILDSELECT                                64   ********
  499  PROC BDY  PARSE                                       1   *
  505  PROC BDY  ERASE                                     398   ****************************************************
  525  PROC BDY  EXPAND                                    254   **********************************
  533  FUNC BDY  MATCH                                     254   **********************************

| LL_SUPPORT:BODY
| 27-JAN-1992 10:37:08
   97  FUNC BDY  ALTERNATE                                  14   **
  103  FUNC BDY  MERGE_RANGES                                1   *
                 ...
 1287  FUNC BDY  OPTION                                      1   *
 1304  FUNC BDY  REPEAT                                      1   *
 1322  PROC BDY  STORE_PATTERN                               6   *
```

Figure 11-4. AdaQuest Unit Coverage Report

```
----------------------------------------------------------------------------------------------------------------------------
29-JAN-1992 13:00                                ADAQUEST BRANCH COVERAGE DETAIL REPORT                                PAGE    1
Library: ADALEX;WORK
       |                                     |             |          Number of Hits - Normalized to Maximum        |      |
Branch |                                     |   Number    | 0       20        40        60        80        100 |      | Branch
Number | Branch Kind                  Line # |   of Hits   | |---------|---------|---------|---------|---------| | F/N  | Number
============================================================================================================================

----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
                                             - - - TEST RUN INFORMATION - - -
Trace File    : USR:[ADATEST]ADAQUEST_RUN_1.ETF;1
Time of Run   : 27-JAN-1992 12:08:39
# Test Cases  : 1

                                             - - - REPORT QUALIFIERS - - -
By Test Case           : No
Test History Included  : Yes
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
                                                          ...
----------------------------------------------------------------------------------------------------------------------------
29-JAN-1992 13:00                                ADAQUEST BRANCH COVERAGE DETAIL REPORT                                PAGE    3
Library: ADALEX;WORK
       |                                     |             |          Number of Hits - Normalized to Maximum        |      |
Branch |                                     |   Number    | 0       20        40        60        80        100 |      | Branch
Number | Branch Kind                  Line # |   of Hits   | |---------|---------|---------|---------|---------| | F/N  | Number
============================================================================================================================
----------------------------------------------------------------------------------------------------------------------------
Program Unit : LLFIND
At Line      : 140
In Comp Unit : LL_COMPILE:BODY                                                                           27-JAN-1992 10:35:12
----------------------------------------------------------------------------------------------------------------------------
    1  | PROGRAM UNIT START             144 |         198 | ***********                                         |      |    1
    2  | LOOP TEST FAIL                 147 |          63 | ****                                                |      |    2
    3  | LOOP TEST PASS                 147 |         898 | **************************************************  |      |    3
    4  | IF TEST PASS                   149 |         429 | ************************                            |      |    4
    5  | ELSIF TEST PASS                151 |         135 | ********                                            |      |    5
    6  | IF TEST PASS                   152 |         135 | ********                                            |      |    6
    7  | ELSE PART                      154 |           0 |                                                     |      |    7
    8  | ELSE PART                      157 |         334 | *******************                                 |      |    8
----------------------------------------------------------------------------------------------------------------------------
                              THIS TEST RUN    CUMULATIVE |                                                     |      |
Unit Invocations                        198           593 |                                                     |      |
Branches Hit                              7             8 |                                                     |      |
Branches in Unit                          8             8 |                             KEY:   First Time Hit   |  F   |
Per Cent Coverage                     87.5%        100.0% |                                    Never Hit        |--N--|
----------------------------------------------------------------------------------------------------------------------------
                                                          ...
```

Figure 11-5. AdaQuest Branch Coverage Detail Report

27-JAN-1992 12:15 ADAQUEST BRANCH COVERAGE SUMMARY REPORT PAGE 1
Library: ADALEX;WORK

- - - TEST RUN INFORMATION - - -

Trace File : USR:[ADATEST]ADAQUEST.ETF;16
Time of Run : 27-JAN-1992 12:08:39
Name : EXAMPLE_1
Description : Run with 1st test file
Test Cases : 1

- - - REPORT QUALIFIERS - - -

By Test Case : No
Test History Included : Yes

27-JAN-1992 12:15 ADAQUEST BRANCH COVERAGE SUMMARY REPORT PAGE 2
Library: ADALEX;WORK

		THIS TEST RUN			CUMULATIVE			
Program Unit	Line #	Branches In Unit	Unit Invocations	Branches Hit	PER CENT COVERAGE	Unit Invocations	--Branches Hit-- New	Total	PER CENT COVERAGE
COVERAGE FOR COMP UNIT : LL_COMPILE:BODY									27-JAN-1992 10:35:12
LL_COMPILE	26	1	1	1	100.0%	1	1	1	100.0%
LLFIND	140	8	198	7	87.5%	198	7	7	87.5%
LLNEXTTOKEN	337	3	134	3	100.0%	134	3	3	100.0%
LLMAIN	352	1	1	1	100.0%	1	1	1	100.0%
READGRAM	383	11	1	11	100.0%	1	11	11	100.0%
BUILDRIGHT	389	15	64	13	86.7%	64	13	13	86.7%
BUILDSELECT	449	3	64	3	100.0%	64	3	3	100.0%
PARSE	499	17	1	11	64.7%	1	11	11	64.7%
ERASE	505	5	398	5	100.0%	398	5	5	100.0%
EXPAND	525	13	254	11	84.6%	254	11	11	84.6%
MATCH	533	7	254	5	71.4%	254	5	5	71.4%
COVERAGE FOR COMP UNIT : LL_SUPPORT:BODY									27-JAN-1992 10:37:08
ALTERNATE	97	17	14	7	41.2%	14	7	7	41.2%
MERGE_RANGES	103	3	1	3	100.0%	1	3	3	100.0%
...									
OPTION	1287	4	1	2	50.0%	1	2	2	50.0%
REPEAT	1304	4	1	2	50.0%	1	2	2	50.0%
STORE_PATTERN	1322	10	6	5	50.0%	6	5	5	50.0%

Figure 11-6. AdaQuest Branch Coverage Summary Report

```
31-JAN-1992 10:02                          ADAQUEST BRANCH COVERAGE SUMMARY REPORT                                PAGE    1
Library: ADALEX_PROBE;WORK               |          |  T H I S   T E S T   C A S E  |               C U M U L A T I V E
                                         | Branches |   Unit      | Branches | PER CENT |    Unit     | --Branches Hit-- | PER CENT
Program Unit                      Line # | In Unit  | Invocations |   Hit    | COVERAGE | Invocations |  New     Total   | COVERAGE
=============================================================================================================================
-----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------
                                              - - - TEST RUN INFORMATION - - -
Trace File   : USR:[ADATEST]ADAQUEST.ETF;18
Time of Run  : 30-JAN-1992 15:01:56
# Test Cases : 175
                                              - - - REPORT QUALIFIERS - - -
By Test Case          : Yes
Test History Included : Yes
-----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------
=============================================================================================================================
31-JAN-1992 10:02                          ADAQUEST BRANCH COVERAGE SUMMARY REPORT                                PAGE    2
Library: ADALEX_PROBE;WORK                          |             |  T H I S   T E S T   C A S E  |               C U M U L A T I V E
                                         | Branches |   Unit      | Branches | PER CENT |    Unit     | --Branches Hit-- | PER CENT
Program Unit                      Line # | In Unit  | Invocations |   Hit    | COVERAGE | Invocations |  New     Total   | COVERAGE
=============================================================================================================================
TC1**************************************************************************************************************************
***********                                          T E S T   C A S E   # 1                                     ***********
*********** Ending at Statement : 398                                                                            ***********
*********** Of Compilation Unit : LL_COMPILE:BODY                                                                ***********
*****************************************************************************************************************************
-----------------------------------------------------------------------------------------------------------------------------
COVERAGE FOR COMP UNIT : LL_COMPILE:BODY                                                          30-JAN-1992 13:26:04
-----------------------------------------------------------------------------------------------------------------------------
LL_COMPILE                          26 |     1    |         1          1     100.0% |        1         1        1      100.0%
LLMAIN                             352 |     1    |         1          1     100.0% |        1         1        1      100.0%
READGRAM                           383 |    11    |         1          8      72.7% |        1         8        8       72.7%
BUILDRIGHT                         389 |    15    |         1          3      20.0% |        1         3        3       20.0%
=============================================================================================================================
31-JAN-1992 10:02                          ADAQUEST BRANCH COVERAGE SUMMARY REPORT                                PAGE    3
Library: ADALEX_PROBE;WORK                          |             |  T H I S   T E S T   C A S E  |               C U M U L A T I V E
                                         | Branches |   Unit      | Branches | PER CENT |    Unit     | --Branches Hit-- | PER CENT
Program Unit                      Line # | In Unit  | Invocations |   Hit    | COVERAGE | Invocations |  New     Total   | COVERAGE
=============================================================================================================================
TC2**************************************************************************************************************************
***********                                          T E S T   C A S E   # 2                                     ***********
*********** Ending at Statement : 398                                                                            ***********
*********** Of Compilation Unit : LL_COMPILE:BODY                                                                ***********
*****************************************************************************************************************************
-----------------------------------------------------------------------------------------------------------------------------
COVERAGE FOR COMP UNIT : LL_COMPILE:BODY                                                          30-JAN-1992 13:26:04
-----------------------------------------------------------------------------------------------------------------------------
BUILDRIGHT                         389 |    15    |         0          5      33.3% |        1         3        6       40.0%
                                                                                ...
```

Figure 11-7. AdaQuest Branch Coverage Report Showing Test Runs

```
mod_cov_nothit_run1        Tue Jan 28 10:48:55 1992        1

27-JAN-1992 12:16                ADAQUEST BRANCH COVERAGE NOT-HIT REPORT                PAGE    1
Library: ADALEX;WORK

                                  - - - TEST RUN INFORMATION - - -
Trace File    : USR:[ADATEST]ADAQUEST.ETF;16
Time of Run   : 27-JAN-1992 12:08:39
Name          : EXAMPLE_1
Description   : Run with 1st test file
# Test Cases : 1

                                  - - - REPORT QUALIFIERS - - -
By Test Case          : No
Test History Included : Yes

27-JAN-1992 12:16                ADAQUEST BRANCH COVERAGE NOT-HIT REPORT                PAGE    2
Library: ADALEX;WORK

                                  |  C U M U L A T I V E  |
                                  |  Per Cent  |  Branches  |
Program Unit                Line # |  Coverage  |  Not Hit   |  IDs of Branches Not Hit
-------------------------------------------------------------------------------------------
COVERAGE FOR COMP UNIT : LL_COMPILE:BODY                                  27-JAN-1992 10:35:12
-------------------------------------------------------------------------------------------
                                  |            |            |
LL_COMPILE                     26 |    100.0%  |          0 |
LLFIND                        140 |     87.5%  |          1 |     7
LLNEXTTOKEN                   337 |    100.0%  |          0 |
LLMAIN                        352 |    100.0%  |          0 |
READGRAM                      383 |    100.0%  |          0 |
BUILDRIGHT                    389 |     86.7%  |          2 |    57    62
BUILDSELECT                   449 |    100.0%  |          0 |
PARSE                         499 |     64.7%  |          6 |   129   130   131   132   133   139
ERASE                         505 |    100.0%  |          0 |
EXPAND                        525 |     84.6%  |          2 |    94   101
MATCH                         533 |     71.4%  |          2 |    83    88
                                  |            |            |
-------------------------------------------------------------------------------------------
COVERAGE FOR COMP UNIT : LL_SUPPORT:BODY                                  27-JAN-1992 10:37:08
-------------------------------------------------------------------------------------------
                                  |            |            |
ALTERNATE                      97 |     41.2%  |         10 |     6     8     9    10    11    12    13    17    18    19
MERGE_RANGES                  103 |    100.0%  |          0 |
LOOK_UP_PATTERN              1273 |     80.0%  |          1 |   288
OPTION                       1287 |     50.0%  |          2 |   294   295
REPEAT                       1304 |     50.0%  |          2 |   298   299
STORE_PATTERN                1322 |     50.0%  |          5 |   301   305   306   307   308
                                  ...
```

Figure 11-8. AdaQuest Branch Coverage Not-Hit Report

27-JAN-1992 12:44 ADAQUEST COVERAGE HISTORY DETAIL REPORT PAGE 1
Library: ADALEX;WORK Comp Unit: LL_COMPILE:BODY
27-JAN-1992 10:35:12

Program Unit	Line #	Number of Branches	THIS TEST RUN: Number of Executions	Branches Hit	PER CENT COVERAGE	CUMULATIVE: Number of Executions	--Branches Hit-- New	Total	PER CENT COVERAGE
TR1			TEST RUN # 1						
Time of Execution : 27-JAN-1992 12:08:39									
Trace File : USR:[ADATEST]ADAQUEST_RUN_1.ETF;1									
LL_COMPILE	26	1	1	1	100.0%	1	1	1	100.0%
LLFIND	140	8	198	7	87.5%	198	7	7	87.5%
LLNEXTTOKEN	337	3	134	3	100.0%	134	3	3	100.0%
LLMAIN	352	1	1	1	100.0%	1	1	1	100.0%
READGRAM	383	11	1	11	100.0%	1	11	11	100.0%
BUILDRIGHT	389	15	64	13	86.7%	64	13	13	86.7%
BUILDSELECT	449	3	64	3	100.0%	64	3	3	100.0%
PARSE	499	17	1	11	64.7%	1	11	11	64.7%
ERASE	505	5	398	5	100.0%	398	5	5	100.0%
EXPAND	525	13	254	11	84.6%	254	11	11	84.6%
MATCH	533	7	254	5	71.4%	254	5	5	71.4%
TR2			TEST RUN # 2						
Time of Execution : 27-JAN-1992 12:21:08									
Trace File : USR:[ADATEST]ADAQUEST.ETF;16									
LL_COMPILE	26	1	1	1	100.0%	2	0	1	100.0%
LLFIND	140	8	197	8	100.0%	395	1	8	100.0%
LLNEXTTOKEN	337	3	136	3	100.0%	270	0	3	100.0%
LLMAIN	352	1	1	1	100.0%	2	0	1	100.0%
READGRAM	383	11	1	11	100.0%	2	0	11	100.0%
BUILDRIGHT	389	15	64	13	86.7%	128	0	13	86.7%
BUILDSELECT	449	3	64	3	100.0%	128	0	3	100.0%
PARSE	499	17	1	11	64.7%	2	0	11	64.7%
ERASE	505	5	406	5	100.0%	804	0	5	100.0%
EXPAND	525	13	260	11	84.6%	514	0	11	84.6%
MATCH	533	7	260	5	71.4%	514	0	5	71.4%

27-JAN-1992 12:44 ADAQUEST COVERAGE HISTORY DETAIL REPORT PAGE 2
Library: ADALEX;WORK Comp Unit: LL_COMPILE.LL_TOKENS
27-JAN-1992 10:39:35

Program Unit	Line #	Number of Branches	THIS TEST RUN: Number of Executions	Branches Hit	PER CENT COVERAGE	CUMULATIVE: Number of Executions	--Branches Hit-- New	Total	PER CENT COVERAGE

*** The test history for this unit is empty ***
...

Figure 11-9. AdaQuest Coverage History Detail Report

27-JAN-1992 12:43 ADAQUEST COVERAGE HISTORY SUMMARY REPORT PAGE 1
Library: ADALEX;WORK

Program Unit	Line #	# Branches Hit	# Branches In Unit	PER CENT COVERAGE	List of Branches Not Hit
Coverage for Comp Unit : LL_COMPILE:BODY					27-JAN-1992 10:35:12
LL_COMPILE	26	1	1	100.0%	
LLFIND	140	8	8	100.0%	
LLPRTSTRING	165	0	5	0.0%	9 10 11 12 13
LLPRTSTRING	165	0	5	0.0%	9 10 11 12 13
LLPRTTOKEN	177	0	3	0.0%	14 15 16
LLSKIPTOKEN	190	0	1	0.0%	17
LLSKIPMODE	203	0	1	0.0%	18
LLSKIPBOTH	217	0	1	0.0%	19
...					
*** Totals ***		72	142	50.7%	
Coverage for Comp Unit : LL_COMPILE.LL_TOKENS					27-JAN-1992 10:39:35
ADVANCE	49	0	12	0.0%	45 46 47 48 49 50 51 52
					53 54 55 56
GET_CHAR	56	0	4	0.0%	1 2 3 4
CHAR_ADVANCE	69	0	4	0.0%	5 6 7 8
...					
*** Totals ***		0	56	0.0%	
Coverage for Comp Unit : LL_COMPILE.LLTAKEACTION					27-JAN-1992 10:34:11
LLTAKEACTION	25	0	69	0.0%	1 2 3 4 5 6 7 8
					9 10 11 12 13 14 15 16
					17 18 19 20 21 22 23 24
...					
*** Totals ***		0	69	0.0%	
Coverage for Comp Unit : LL_SUPPORT:BODY					27-JAN-1992 10:37:08
ALTERNATE	97	14	17	82.4%	6 12 17
MERGE_RANGES	103	3	3	100.0%	
...					
LOOK_AHEAD	1265	0	1	0.0%	286
LOOK_UP_PATTERN	1273	4	5	80.0%	288
OPTION	1287	2	4	50.0%	294 295
REPEAT	1304	2	4	50.0%	298 299
STORE_PATTERN	1322	5	10	50.0%	301 305 306 307 308
*** Totals ***		193	309	62.5%	

Figure 11-10. AdaQuest Coverage History Summary Report

```
================================================================================
08-MAR-1992 12:01          ADAQUEST TEST RUN REPORT                  PAGE    1


================================================================================
--------------------------------------------------------------------------------
Trace File       : USR:[ADATEST]SUN_RUN4.ETF;1
Time of Run      : 08-MAR-1992 11:35:06
Name             : TIMING_RUN_1
Description      : Sample run for timing data using test file 1
# of Test Cases : 1
--------------------------------------------------------------------------------

Test Run Units :

LL_COMPILE:BODY
   Instrumented From File :
      USR:[ADATEST]LL_COMPILE.ADA;1
   Instrumentation Parameters :
      Timing = CPU
      Timing Intervals (Start/Stop Source Line Numbers) :
         394 - 446
         453 - 459
         463 - 496
         653 - 663
         667 - 717
         721 - 723
         727 - 730

LL_SUPPORT:BODY
   Instrumented From File :
      USR:[ADATEST.ADALEX2]LL_SUP_BODY.ADA;1
   Instrumentation Parameters :
      Timing = CPU
      Timing Intervals (Start/Stop Source Line Numbers) :
         220 - 283
         291 - 351
         365 - 460
         465 - 501
         508 - 546
```

Figure 11-11. AdaQuest Interval Test Run Report

```
08-MAR-1992 12:02        ADAQUEST INTERVAL TIMING REPORT                PAGE    1

           Start / Stop      Number of    Minimum Time   Maximum Time   Average Time
Interval    Line Number     Executions     hh:mm:ss.cc    hh:mm:ss.cc    hh:mm:ss.cc
```

- - - TEST RUN INFORMATION - - -

```
Trace File    : USR:[ADATEST]SUN_RUN4.ETF;1
Time of Run   : 08-MAR-1992 11:35:06
Name          : TIMING_RUN_1
Description   : Sample run for timing data using test file 1
# Test Cases  : 1
```

- - - REPORT QUALIFIERS - - -

```
By Test Case         : No
```

```
08-MAR-1992 12:02        ADAQUEST INTERVAL TIMING REPORT                PAGE    2

           Start / Stop      Number of    Minimum Time   Maximum Time   Average Time
Interval    Line Number     Executions     hh:mm:ss.cc    hh:mm:ss.cc    hh:mm:ss.cc
-------------------------------------------------------------------------------------
CPU TIMING FOR COMP UNIT : LL_COMPILE:BODY
-------------------------------------------------------------------------------------
    1        394 / 446            64      00:00:00.00    00:00:00.01    00:00:00.00
    2        453 / 459            64      00:00:00.00    00:00:00.01    00:00:00.00
    3        463 / 496             1      00:00:00.34    00:00:00.34    00:00:00.34
    5        667 / 717             1      00:00:00.43    00:00:00.43    00:00:00.43
    6        721 / 723             1      00:00:00.78    00:00:00.78    00:00:00.78
    7        727 / 730             1      00:00:00.82    00:00:00.82    00:00:00.82
-------------------------------------------------------------------------------------
CPU TIMING FOR COMP UNIT : LL_SUPPORT:BODY
-------------------------------------------------------------------------------------
    4        465 / 501             4      00:00:00.01    00:00:00.01    00:00:00.01
    5        508 / 546             3      00:00:00.00    00:00:00.00    00:00:00.00
```

Figure 11-12. AdaQuest Interval Timing Report

12. AutoFlow-Ada

AutoFlow-Ada generates flowcharts from Ada source code. These flowcharts can be used to help understand an Ada program and to document it. Versions of AutoFlow that operate on C, Cobol, Fortran, and Pascal code are also available. In addition to flowcharts, these other versions generate structure charts and can interface with the KnowledgeWare/ ADW and the Texas Instruments IEW CASE systems via an import file. The C version also includes the capability to instrument source code to report on test coverage at the branch level; results can then be automatically annotated on flow charts.

12.1 Tool Overview

AutoFlow was developed by AutoCASE Technology. The AutoFlow family as a whole has over 3,000 users. The first Ada version of this product was released early in 1992 and has over 10 users. It runs on IBM PC machines under DOS (version 3.0 or higher) and OS/ 2. The evaluation was performed on version 1.02 of AutoFlow-Ada. At the time of evaluation, the price for AutoFlow was $9,950.

AutoFlow-Ada generates self-explanatory block-structured flowcharts using a flowchart layout copyrighted by AutoCASE Technology. It is intended for use on programs with correct Ada syntax, that is, compilable programs. Compiler directives are treated as comments and not expanded. Consequently, in some circumstances, it may be necessary to use the fully expanded preprocessed listing file provided by many Ada compilers as input to AutoFlow-Ada. The tool can be used in interactive or batch mode. In interactive mode it allows the user to both create and browse flowcharts, selectively saving or printing chosen charts. In batch mode, all produced flowcharts are automatically saved to disk.

Usually one flowchart is generated for each Ada procedure. Some of these flowcharts may be very large and various options are provided for dealing with flowcharts that cannot fit on a single page. The best of these is a block-structured page-break algorithm that uses a top-down refinement approach to break a large flowchart into subcharts that can be presented on separate pages. Additional flexibility is provided by allowing the user to specify the size of page used. Another option is to limit the size of the box in which flowcharts are presented. In this case, flowchart elements that do not fit into the specified box are represented by a string of dots. Alternatively, the user can request that a flowchart saved to disk is divided into strips that can be manually combined to make a large chart.

12.2 Observations

Ease of use. The installation and operation of AutoFlow-Ada is straightforward. The tool is fast: the documentation cites an example of generating flowcharts for an Ada program in excess of ten thousand lines of code, where the average time to generate each flowchart page was less than 0.5 seconds.

AutoFlow-Ada includes a number of special options and utilities that facilitate its use. The utility *mkdoall*, for example, generates command files that will invoke AutoFlow-Ada on multiple source files. Utilities and functions that support its use with non-IBM compatible printers are also provided. Additionally, a file format conversion utility is available to convert ASCII file into PostScript, HPGL, and PIC formats that can be sent to special output devices, or used with desktop publishing software, to prepare high quality documentation.

Documentation and user support. The documentation is sufficient for tool use. AutoCASE Technology provided good support and was helpful and prompt in addressing encountered problems.

Ada restrictions. AutoFlow-Ada supports full Ada, the only restrictions being that each procedure or function is limited to 2,048 basic blocks and that input source lines are limited to 127 characters.

Problems encountered. AutoFlow-Ada ran on the sample Ada Lexical Analyzer Generator source code. Various problems were encountered if the size of generated flowcharts was not constrained or when some particular page sizes were specified. These problems included the process hanging and incorrect referencing between subcharts. AutoCASE corrected the underlying problems and provided a new copy of the tool.

12.3 Planned Additions

Version 2 of AutoFlow-Ada is scheduled for release in the fourth quarter of 1992. It will include the generation of structure charts and a graphical user interface. This version will also be available on major Unix platforms.

12.4 Sample Outputs

Figures 12-1 through 12-6 provide sample outputs from AutoFlow-Ada.

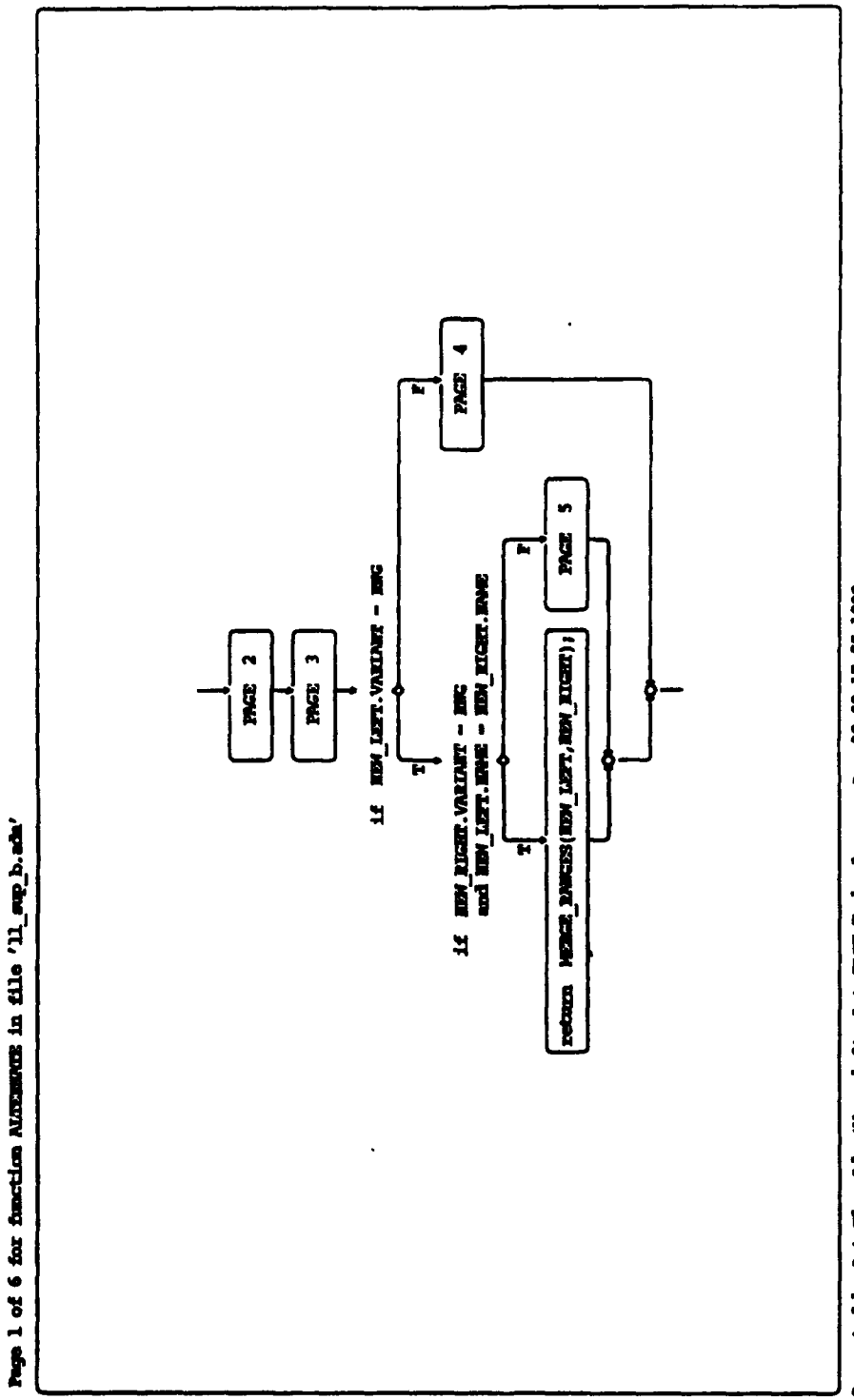


Figure 12-1. AutoFlow-Ada Page 1 of 6 Flowgraph for Function ALTERNATE

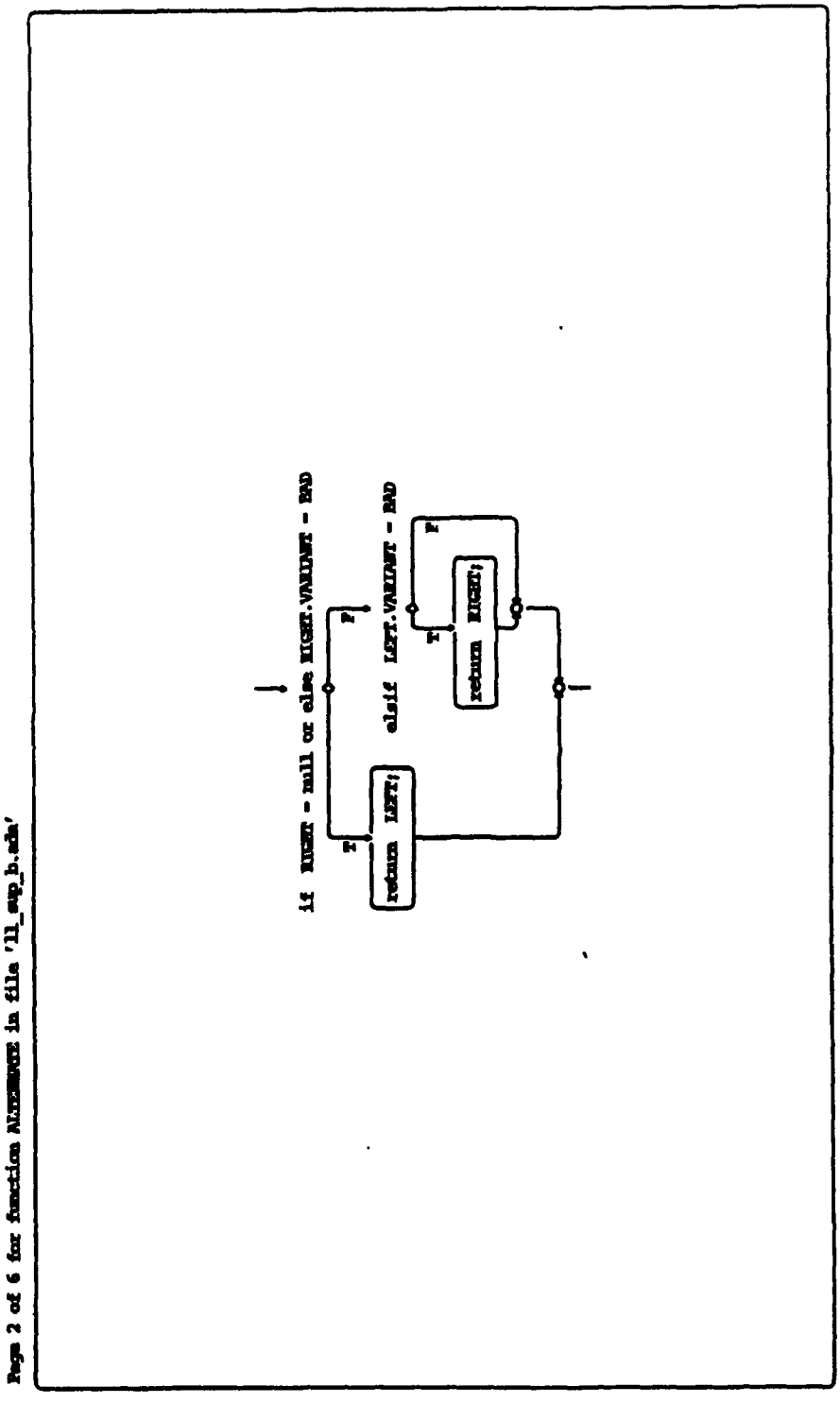


Figure 12-2. AutoFlow-Ada Page 2 of 6 Flowgraph för Function ALTERNATE

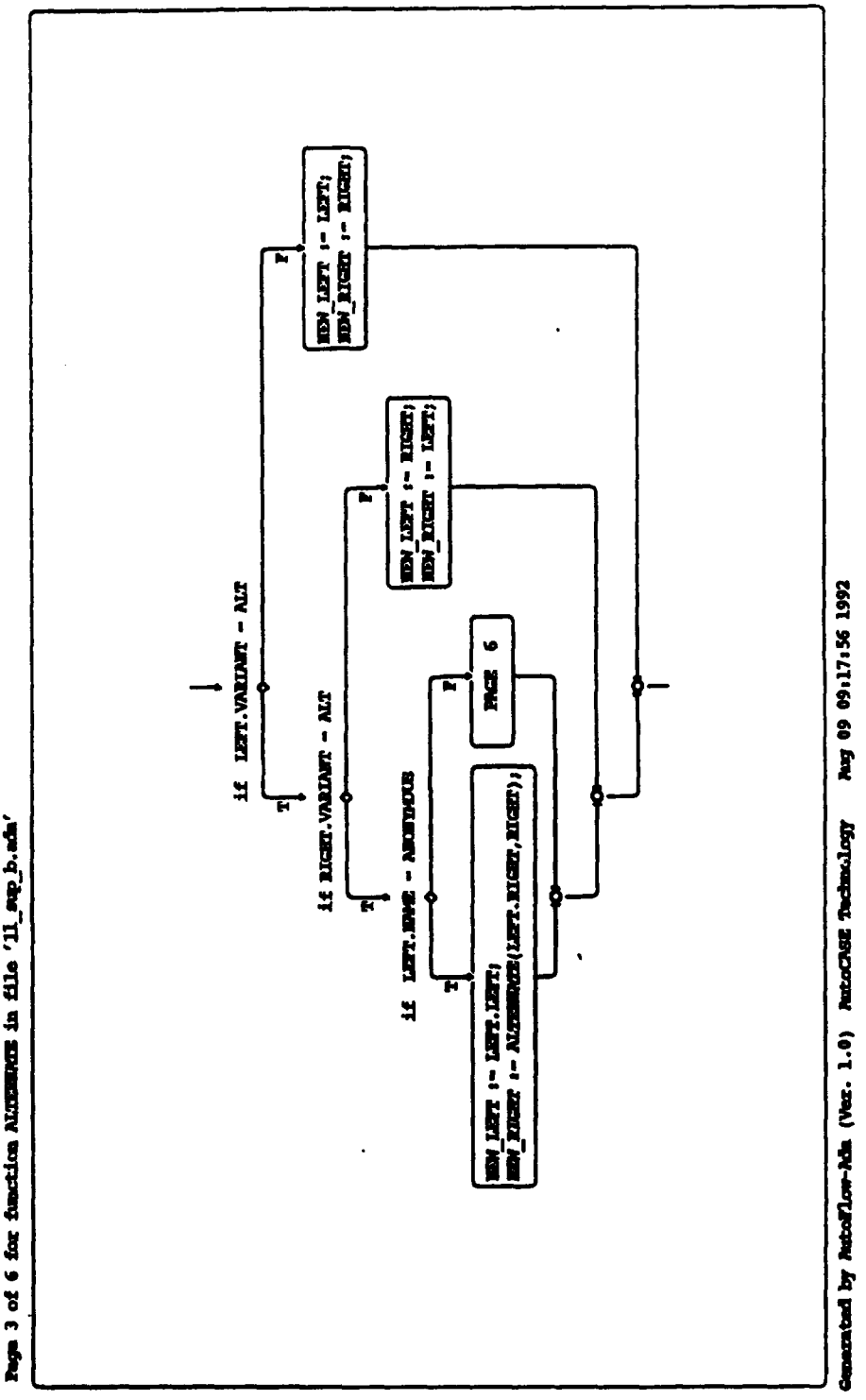


Figure 12-3. AutoFlow-Ada Page 3 of 6 Flowgraph for Function ALTERNATE

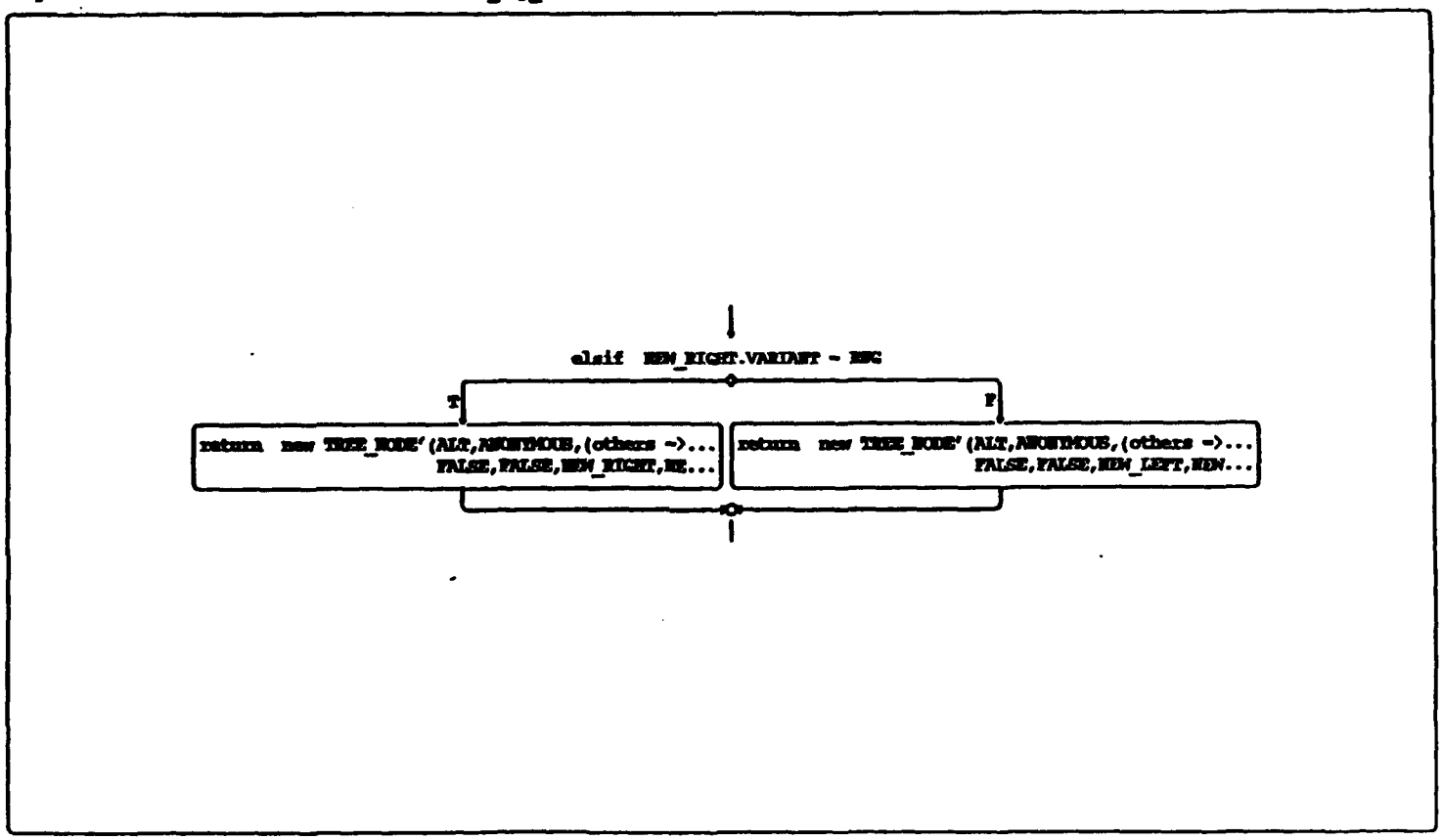


Figure 12-4. AutoFlow-Ada Page 4 of 6 Flowgraph for Function ALTERNATE

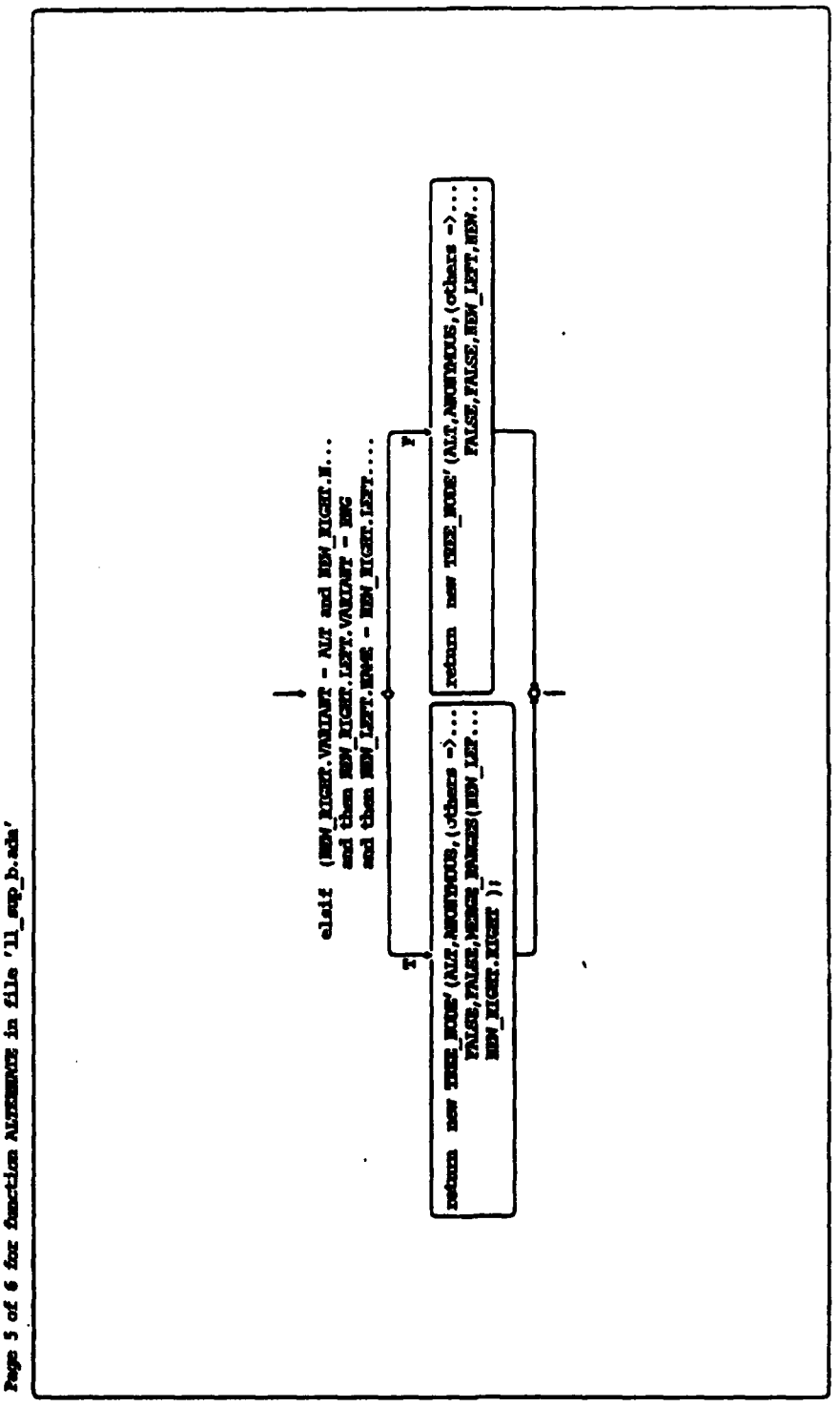


Figure 12-5. AutoFlow-Ada Page 5 of 6 Flowgraph for Function ALTERNATE

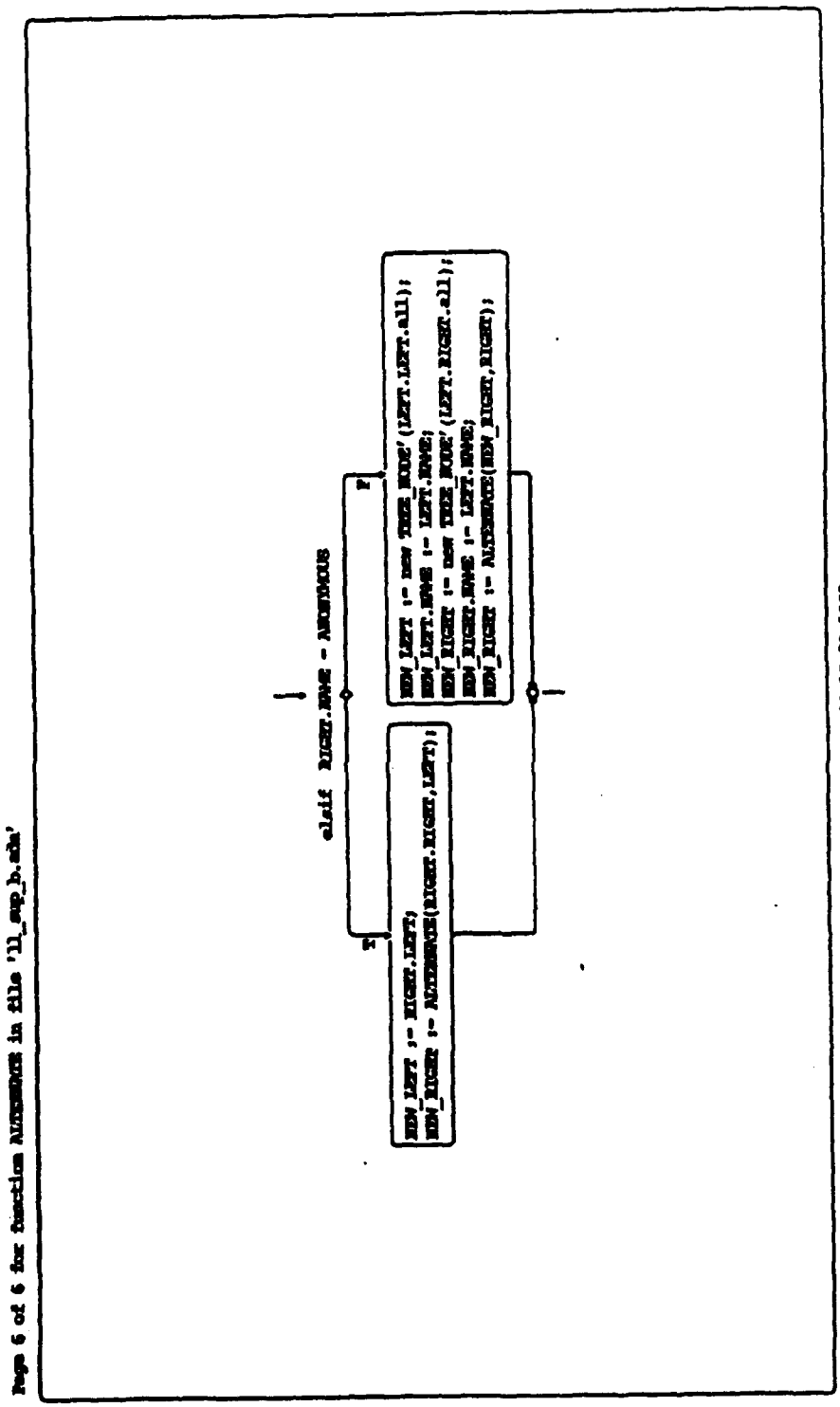


Figure 12-6. AutoFlow-Ada Page 6 of 6 Flowgraph for Function ALTERNATE

13. DISTRIBUTED DEFECT TRACKING SYSTEM (DDTs)

The Distributed Defect Tracking System (DDTs) provides for tracking and managing defects and change requests throughout the life cycle of a software or hardware product. It is designed to support large organizations with multiple sites and so is fully distributed and suitable for use in a heterogeneous network. DDTs supports multiple development teams, allowing data to be maintained for several projects simultaneously. In addition to reporting, searching, and query tools, DDTs informs appropriate users of changes to defect states to provide closed-loop tracking.

13.1 Tool Overview

This product was developed by QualTrak Corporation and has been marketed since 1989. There are over 100 sites using DDTs, with some estimated 5,000 users. QualTrak provides consultancy and training, and supports its product with a hot-line service and an on-line users group. A newsletter is expected to become available in the near future. DDTs is available under SunOS on Sun-3 and Sun-4 systems, under HP-UX on HP-9000, under AIX on IBM RS-6000 and Apollo systems, under Ultrix on DECstations and VAXs, and under SCO Unix. It uses *troff*, *tbl*, *sort*, *awk* Bourne shell Unix utilities, but is DBMS independent. For a local network, it supports network file sharing (NFS), ethernet client/server User Datagram Protocol (UDP), and the Transmission Control Protocol (TCP). Electronic mail is supported for remote networks. The examination was performed on version 2.1.6 of this product running on a Sun-3 system. At the time of evaluation, prices for DDTs started at $6,000.

DDTs groups defects by project to allow reporting on both the defects in a particular project and to support organizational quality assurance activities across projects. Each project is associated with one computer system, known as the *home system*. All of a project's defects reside on that home system and on the submitters' systems as well. In addition, a subscription facility to a project is supported; in this case, defects are maintained on the home system and the subscriber's system. A secure-in dial facility is available that provides easy local access to defect information about remote projects. Closed-loop tracking means that defect submitters are automatically informed of all changes in a defect's status by electronic mail.

DDTs can be used in either a menu-driven or command-driven manner. In the first case, a user has two avenues of access; one provides the full set of functions suitable for a developer, and the other provides a subset of functions tailored towards defect submitters.

The system defines a defect life cycle which allows defects to be managed using a state transition mechanism; both forward and backward transitions are supported. A defect life cycle starts with its submission and, usually, ends with its resolution. There are nine predefined defect states, though the user can define others by including them in a state transition table and defining allowable state transitions. (DDTs warns of any illegal transition attempts.) Defects can be classified as enhancement requests and subsequently tracked by DDTs.

DDTs uses a template to guide user entry of defect reports. Information is grouped into the following areas: detection, submitter, laboratory, resolution, and verification information. Detection information is used to specify, for example, the detection method, the development phase in which the defect was detected, and defect severity (one of five levels) in addition to identification of the test system operating system and affected project. When available, information about the defect submitter is added automatically. The laboratory information captures information pertaining to diagnosing the defect. In addition to identifying the responsible engineer, it records the type and cause of the underlying defect, recommended change, and estimated fix time and date. The resolution information is similar. Again the responsible person is identified, but this time the actual effort required to make the fix, the development phase when the fix was made, and location of actual changes are recorded. Finally, the verification information identifies who accepted the resolution.

Defect reports can be supplemented by *enclosures*. These are additional files containing supplemental ASCII text. They can be used, for example, to include the data files needed to reproduce a problem. There is no limit to the number of enclosures that can be linked to a defect report. DDTs automatically brings up a change editor for creating enclosures. Although the *vi* editor is used by default, the user can request other editors.

DDTs provides several predefined report formats. These conform with the proposed IEEE Standard P-1044, and with DoD-STD-2167A. They include, for example, a list of all unresolved defects for one or more projects and a list of defects in selected states. A number of sorting filters are available for use in constructing specialized report formats. A nice feature is a weekly report program that can be used to produce reports automatically. The metrics provided in weekly reports include such information as the arrival rate, fix rate, number

defects assigned to each project engineer, resolved and unresolved defects, and when these defects were found and/or fixed.

On-line defect report displays are also available. These allow a user to identify all the defects he submitted or the unresolved defects he, or another engineer, is responsible for. The contents of selected defect reports can be displayed with an index pointer used to move between different defect reports. The user can also search this index for a given string. Additional search and query facilities are provided to answer ad hoc questions. The search option allows displaying all unresolved defects and unresolved defects of severity 1 and 2 for one or more projects. It matches a user-defined string against the one-line summaries of defect descriptions kept for each defect. The query function allows the user to specify a search string composed of defect keywords, operators, and values combined in a C-like expression.

DDTs provides explicit support for a number of administration functions. These include cleaning up log files, checking and repairing the database, showing the status of DDTs projects, and managing projects. Setting up a new project involves setting applicable template files and state transition rules. The administrator also specifies the individuals and groups who should be notified of changes to defect states and those who are permitted to change defect states. He can customize DDTs by adding or deleting a defect state, adding or deleting a field in the defect reports, and changing the dialog that occurs with the user when a state transition occurs. In addition to modifying the predefined management reports, the administrator can create new report types. Finally, the administrator is provided with guidance for converting existing defect reports to DDTs format.

13.2 Observations

Ease of use. DDTs recognizes two types of users: defect submitters, and developers who repair defects. While the menu interface provided for each type of user is similar, this distinction allows providing a simpler interface for defect submitters. Context-sensitive help is available in both cases. Additional guidance for expert users is available as a set of tips. These take the form of short excerpts from the on-line manual pages and provide an introduction to the search and query functions. Expert users can also use DDTs through a command interface.

Template file mechanisms provide for customization and specialized defect reports can be defined to augment the predefined reports provided by DDTs. The system includes sev-

eral levels of flexibility. For example, each project can employ different screens, prompting, and states transitions.

The DDTs import facility is a valuable capability. It allows the definition of converters that take existing defect reports in a defined format, convert them to a defined DDTs format, and place them in the DDTs database.

Documentation and user support. DDTs is designed so that it can be used without documentation. Nevertheless, it is well supported by documentation that includes a tutorial, several examples, and sample outputs. Unix-like on-line manual pages provide for quick reference and can be integrated into the on-line manuals supported by a Unix system.

Installation procedures are well described. They include special information that, for example, helps a system administrator determine where to place the product by providing an estimate of the rate of growth of the database, as well as estimates of dynamic storage requirements.

Problems encountered. No problems were encountered in the use of DDTs.

13.3 Recent Changes and Planned Additions

A new product, Remote Distributed Defect Tracking System (RDDTs), released in summer 1992, provides a restricted submit-only version of DDTs.

DDTs release 3.0 is due to be released in December 1992. This version will support an X-11 graphical user interface as well as the existing *tty* interface. It will also support PostScript for enhanced graphical charts.

The QTET test harness is a product under development to provide an interface between DDTs and test execution tools. It is based on the public domain Test Environment Toolkit; QualTrak Corp. has added a graphical user interface and bound the Test Environment Toolkit to DDTs. QTET is expected to become available in the second quarter of 1993.

13.4 Sample Outputs

Figures 13-1 through 13-7 provide sample outputs from DDTs. Figure 13-8 provides an example of the outputs available with the DDTs graphical user interface; it was supplied by QualTrak Corp.

```
********************************************************************************

Bug SFDaa03277                      DDTs                      Submitted 910305
  ASSIGNED defect report       bugs(1), version 2.1           Assigned 910305
                                  2 enclosures

  "Enclosure date stamp is incorrectly updated"

DETECTION INFORMATION                       LABORATORY INFORMATION
 Detection method:  customer use             Assigned engineer:    rico
 Detected in phase: post-release
 Test program name: bugs
 Test system:
 Version of OS:
 Problem severity:  3
 Affects project:   ddts
 Need fix by:       910909
SUBMITTER INFORMATION
 Submitter:      Mike Manley
 Organization:   QT LABX
 Phone number:   33157
 Address:        mikey!mmanley

************ Problem (Added 910305 by mmanley) ***********

From Lori Pope at Pacesetter

>2. (New?) When we attempt to modify an existent enclosure by:

>       1. selecting "m" when viewing the enclosure
>       2. exit the editor without saving the modifications (eg. no
>       modifications were performed) - in vi you would exit with ":q!"
>       3. Exit ddts by typing "q".

>the enclosure's modification date is updated.

>For now, the work around is to exit ddts by typing "x" instead of
>"q" - this exits ddts without saving any changes.
>That may not be satisfactory if you have made changes in other SWRs
>that you wish to keep.


************ design ideas (Added 910606 by ddts) ***********
stat the file before going to the editor and see if the time changed
```

Figure 13-1. DDTs Sample Defect Report

DDTS MANAGEMENT SUMMARY
of
DEFECTS by PROJECT by STATE
(Tue Jul 14 14:19:11 EDT 1992)

Project	New	Assnd	Open	Rslvd	Verif	Dup	Postp	Total
DDTs	1	18	0	84	0	0	0	103
TOTAL	1	18	0	84	0	0	0	103

Youngest Bug Date => 911221
Oldest Bug Date => 901029
Software Versions => 2.1 3.0 2.1.3
unk 2.2 2.0.3
2.1.2 2.0 2.0.1
bar 1.0
O.S. Versions => 4.1 4.0 Sun
SunOS unk a
none any 3.5

DDTS MANAGEMENT SUMMARY
of
DEFECTS by PROJECT by SEVERITY
RESOLVED & UNRESOLVED BUGS
(Tue Jul 14 14:19:16 EDT 1992)

Project	Sev1	Sev2	Sev3	Sev4	Sev5	Total
DDTs	10	14	65	12	2	103
TOTAL	10	14	65	12	2	103

Youngest Bug Date => 911221
Oldest Bug Date => 901029
Software Versions => 2.1 3.0 2.1.3
unk 2.2 2.0.3
2.1.2 2.0 2.0.1
bar 1.0
O.S. Versions => 4.1 4.0 Sun
SunOS unk a
none any 3.5

Figure 13-2. DDTs Management Summary Report: Defect Reports

DDTS MANAGEMENT SUMMARY
of
DEFECTS by ENGINEER by SEVERITY
UNRESOLVED DEFECTS ONLY
(Tue Jul 14 14:19:22 EDT 1992)

Assigned Engineer	Sev 1	Sev 2	Sev 3	Sev 4	Sev 5	Total
mmanley	0	0	4	0	0	4
rico	0	1	9	4	1	15
david	0	0	0	0	0	0
davep	0	0	0	0	0	0
carol	0	0	0	0	0	0
UNASSIGNED	0	0	0	0	0	0
TOTAL	0	1	13	4	1	19

```
Projects surveyed =>  DDTs
Youngest Bug Date =>  911221
Oldest Bug Date   =>  901029
Software Versions =>  2.1  3.0  2.1.3
                      unk  2.2  2.0.3
                      2.1.2  2.0  2.0.1
                      bar  1.0
O.S. Versions     =>  4.1  4.0  Sun
                      SunOS  unk  a
                      none  any  3.5
```

Figure13-2 continued: DDTs Management Summary Report: Defect Reports

DDTS MANAGEMENT SUMMARY
of
DEFECTS by SUBMITTING ENGINEER by SEVERITY
(Tue Jul 14 14:19:35 EDT 1992)

Submitting Engineer	Sev 1	Sev 2	Sev 3	Sev 4	Sev 5	Total
mmanley	4	3	29	3	1	40
cindy	3	5	22	7	1	38
ddts	2	6	11	2	0	21
carol	0	0	3	0	0	3
rico	1	0	0	0	0	1
UNASSIGNED	0	0	0	0	0	0
TOTAL	10	14	65	12	2	103

```
Projects surveyed =>  DDTs
Youngest Bug Date =>  911221
Oldest Bug Date   =>  901029
Software Versions =>  2.1  3.0  2.1.3
                      unk  2.2  2.0.3
                      2.1.2  2.0  2.0.1
                      bar  1.0
O.S. Versions     =>  4.1  4.0  Sun
                      SunOS  unk  a
                      none  any  3.5
```

Figure13-2 continued: DDTs Management Summary Report: Defect Reports

DDTS MANAGEMENT SUMMARY
DEFECT ARRIVAL & REPAIR RATE
ALL SEVERITY LEVELS
(Tue Jul 14 14:19:40 EDT 1992)

Week	Date	# New	# Resolved	Diff	# Unresolved
1	901028	1	0	1	2
2	901104	0	0	0	2
3	901111	3	2	1	3
4	901118	0	0	0	3
5	901125	0	0	0	3
6	901202	0	0	0	3
7	901209	2	1	1	4
8	901216	1	0	1	5
9	901223	0	0	0	5
10	901230	0	0	0	5
11	910106	1	0	1	6
12	910113	4	0	4	10
13	910120	1	0	1	11
14	910127	4	0	4	15
15	910203	0	0	0	15
16	910210	1	0	1	16
17	910217	0	0	0	16
18	910224	0	0	0	16
19	910303	6	0	6	22
20	910310	0	0	0	22
21	910317	1	0	1	23
22	910324	2	10	-8	15
23	910331	0	0	0	15
		...			
59	911208	3	2	1	19

Projects surveyed => DDTs
Youngest Bug Date => 911221
Oldest Bug Date => 901029

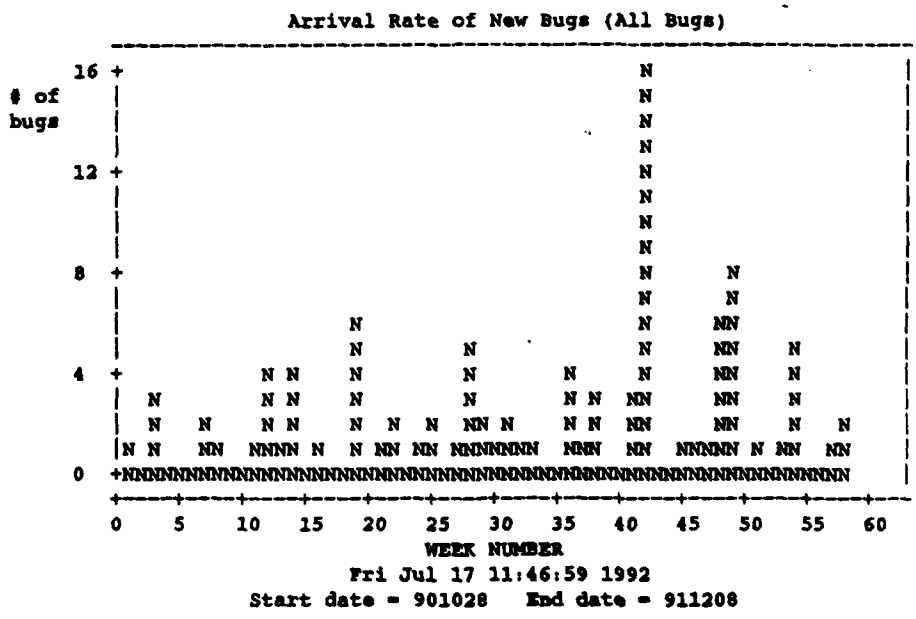
Figure 13-3. DDTs Management Summary Report: Defect Arrival and Repair Rate (All Levels)

DDTS MANAGEMENT SUMMARY
DEFECT ARRIVAL & REPAIR RATE
SEVERITY 1 & 2 DEFECTS ONLY
(Tue Jul 14 14:19:40 EDT 1992)

Week	Date	# New	# Resolved	Diff	# Unresolved
1	901028	0	0	0	0
2	901104	0	0	0	0
3	901111	0	0	0	0
4	901118	0	0	0	0
5	901125	0	0	0	0
6	901202	0	0	0	0
7	901209	0	0	0	0
8	901216	0	0	0	0
9	901223	0	0	0	0
10	901230	0	0	0	0
11	910106	0	0	0	0
12	910113	0	0	0	0
13	910120	0	0	0	0
14	910127	1	0	1	1
15	910203	0	0	0	1
16	910210	0	0	0	1
17	910217	0	0	0	1
18	910224	0	0	0	1
19	910303	1	0	1	2
20	910310	0	0	0	2
21	910317	0	0	0	2
22	910324	1	3	-2	0
23	910331	0	0	0	0
		...			
59	911208	1	1	0	1

Projects surveyed => DDTs
Youngest Severity 1 or 2 Bug Date => 911213
Oldest Severity 1 or 2 Bug Date => 910131

Figure 13-4. DDTs Management Summary Report: Defect Arrival and Repair Rate (Sev. 1 & 2)



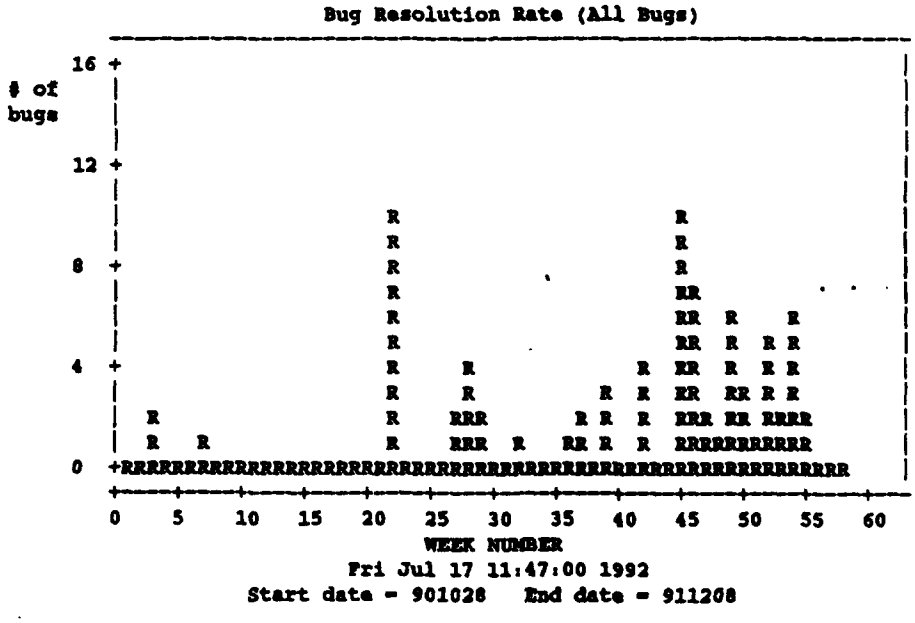


Figure 13-5. DDTs Management Summary Report: Sample Histograms

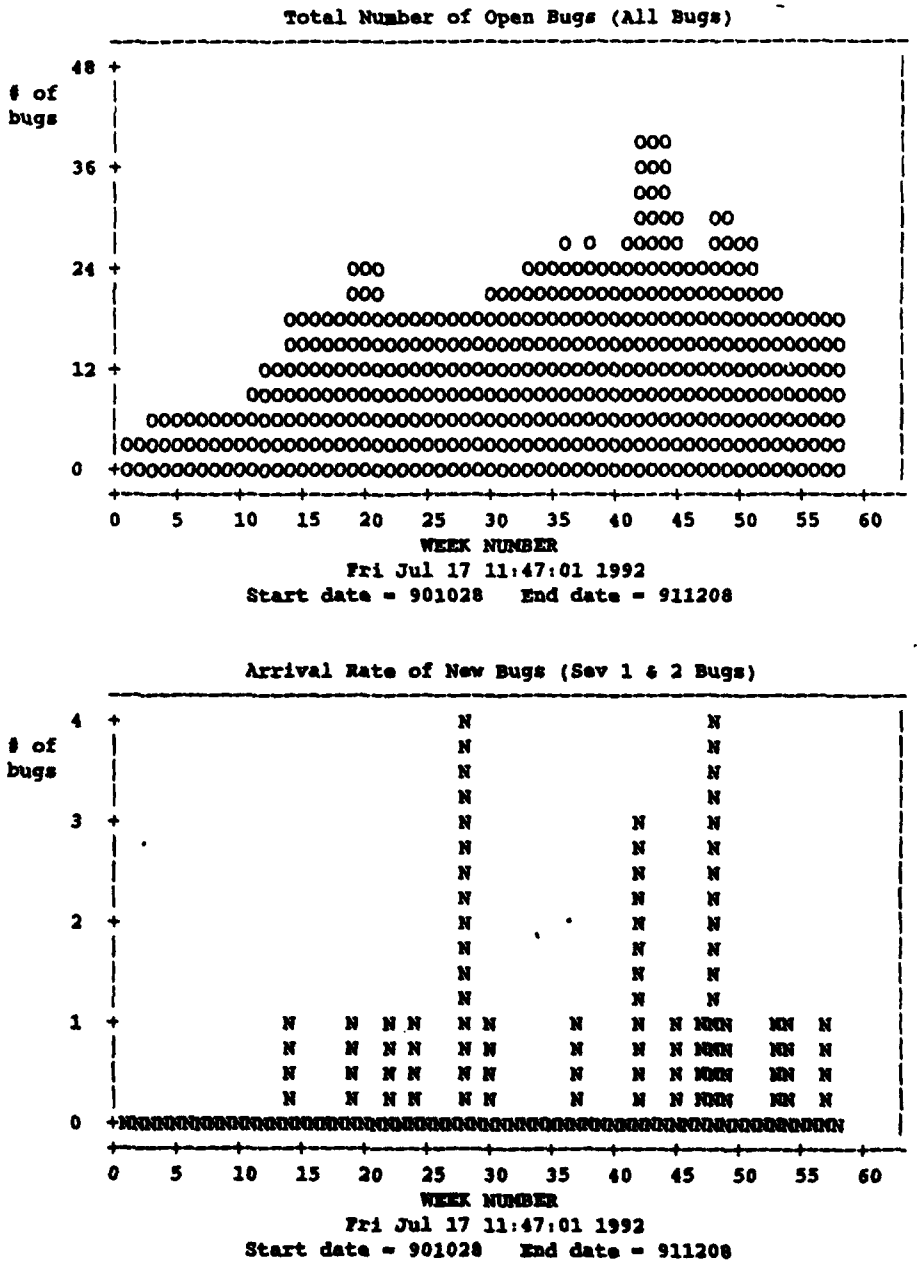
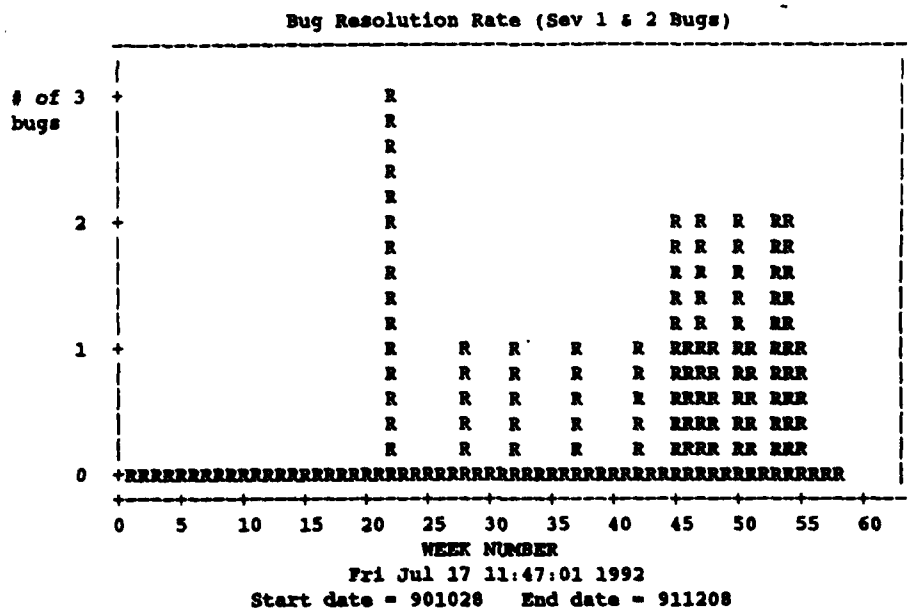


Figure13-5 continued: DDTs Management Summary Report: Sample Histograms



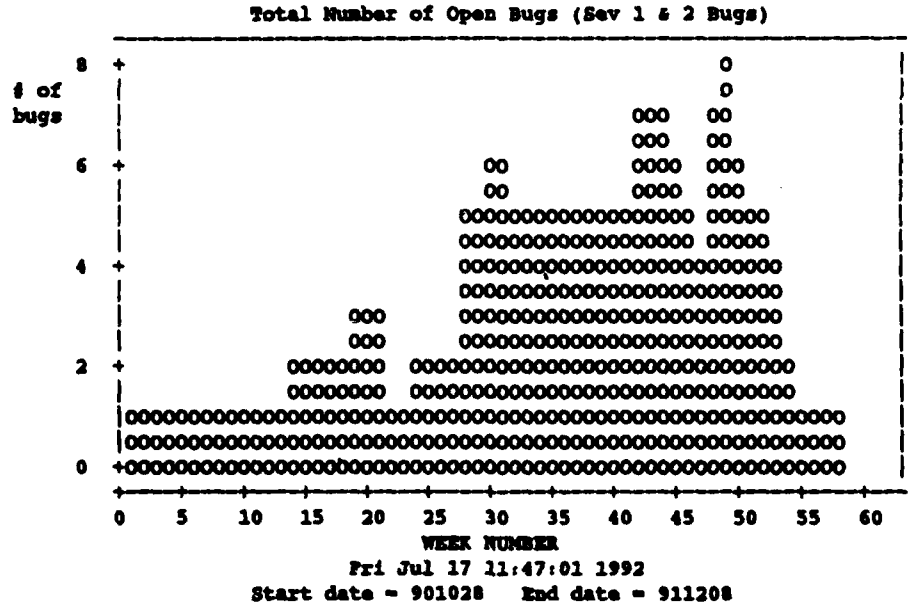


Figure13-5 continued: DDTs Management Summary Report: Sample Histograms

```
                    DDTS MANAGEMENT SUMMARY
                    Three Line Bug Summaries
                  Tue Jul 14 14:20:02 EDT 1992

DEFECTS FOR PROJECT  DDTs

Bug Number = QQQaa00051,   Project = DDTs,
St=N, Sv=3,  Things to remember for the DDTs Installation Upgrade
Module: upgrade, Vers = 2.1,Engr = mmanley,Found: 911019,Fixed: ??

Bug Number = QQQaa00079,   Project = DDTs,
St=A, Sv=2,  bugs(1) index printing needs to be much faster
Module: bugs(1), Vers = 2.1,Engr = rico,Found: 911213,Fixed: ??

Bug Number = QQQaa00026,   Project = DDTs,
St=A, Sv=3,  Only 1 line should be repeated on page forward thru index
Module: bugs(1), Vers = 2.1,Engr = rico,Found: 910908,Fixed: ??

Bug Number = QQQaa00075,   Project = DDTs,
St=A, Sv=3,  This is a reminder about malloc(3)
Module: bugs(1), Vers = 3.0,Engr = rico,Found: 911201,Fixed: ??

Bug Number = QQQaa00078,   Project = DDTs,
St=A, Sv=3,  Adminbug needs to set up CM stuff
Module: adminbug, Vers = 2.1,Engr = mmanley,Found: 911212,Fixed: ??

Bug Number = QQQaa00081,   Project = DDTs,
St=A, Sv=3,  the mail.subject template file needs documentation
Module: mail.subject, Vers = 3.0,Engr = rico,Found: 911213,Fixed: ??

Bug Number = QQQaa00086,   Project = DDTs,
St=A, Sv=3,  CM-notify needs to be in proj.notify file
Module: adminbug, Vers = 3.0,Engr = mmanley,Found: 911219,Fixed: ??

Bug Number = SFDaa03265,   Project = DDTs,
St=A, Sv=3,  G.E. suggests moving & per-projecting some template files
Module: bugs(1), Vers = 2.1,Engr = rico,Found: 910114,Fixed: ??

Bug Number = SFDaa03270,   Project = DDTs,
St=A, Sv=3,  G.E. wants to have a mechanism for total mail supression per sta
Module: bugmail, Vers = 2.1,Engr = rico,Found: 910130,Fixed: ??

Bug Number = SFDaa03276,   Project = DDTs,
St=A, Sv=3,  Last-mod not updated when enclosure is modified
Module: bugs(1), Vers = 2.1,Engr = rico,Found: 910305,Fixed: ??
                                    ...

Bug Number = SFDaa03290,   Project = DDTs,
St=R, Sv=5,  New bugs loaded via bbox are not displayed
Module: bugs(1), Vers = 2.1,Engr = rico,Found: 910419,Fixed: 911221
```

Figure 13-6. DDTs Management Summary Report: Bug Summaries

```
                    DDTS MANAGEMENT SUMMARY
                              of
                       General Statistics
                  (Tue Jul 14 14:20:09 EDT 1992)

Projects surveyed =>  DDTs
Youngest Bug Date =>  911221
Oldest Bug Date   =>  901029
Software Versions =>  2.1  3.0  2.1.3
                      unk  2.2  2.0.3
                      2.1.2  2.0  2.0.1
                      bar  1.0
O.S. Versions     =>  4.1  4.0  Sun
                      SunOS  unk  a
                      none  any  3.5


Assigned Engineer Statistics
  Of 103 assigned bugs:
      NO ONE    assigned     0 bugs =>    0.00%
      mmanley   assigned    44 bugs =>   42.72%
      rico      assigned    43 bugs =>   41.75%
      david     assigned    15 bugs =>   14.56%
      carol     assigned     1 bugs =>    0.97%

Bug Submission Statistics
  Of 103 bugs submitted:
      mmanley   submitted   40 bugs =>   38.83%
      cindy     submitted   38 bugs =>   36.89%
      ddts      submitted   21 bugs =>   20.39%
      carol     submitted    3 bugs =>    2.91%
      rico      submitted    1 bugs =>    0.97%

How Found Statistics
  Of 103 bugs found:
     10 bugs found by  author code review      =>    9.71%
     20 bugs found by  in-house normal use     =>   19.42%
      2 bugs found by  group code review       =>    1.94%
     60 bugs found by  customer use            =>   58.25%
      8 bugs found by  interactive test        =>    7.77%
      1 bugs found by  random unplanned test   =>    0.97%
      2 bugs found by  functional test         =>    1.94%

How Resolved Statistics
  Of 84 bugs resolved:
     53 bugs resolved by  source code          =>   63.10%
      4 bugs resolved by  design               =>    4.76%
      6 bugs resolved by  documentation        =>    7.14%
     13 bugs resolved by  no fix               =>   15.48%
      3 bugs resolved by  unreproducible       =>    3.57%
      5 bugs resolved by  not a bug            =>    5.95%

When Caused Statistics
```

Figure 13-7. DDTs Management Summary Report: General Statistics

```
  Of 81 bugs:
     15 bugs caused during  design          =>   18.52%
     36 bugs caused during  post-release    =>   44.44%
     12 bugs caused during  alpha test      =>   14.81%
      3 bugs caused during  beta test       =>    3.70%
     10 bugs caused during  implementation  =>   12.35%
      3 bugs caused during  investigation   =>    3.70%
      2 bugs caused during  integration     =>    2.47%

When Found Statistics
  Of 103 bugs found:
     12 bugs found during  alpha test       =>   11.65%
     74 bugs found during  post-release     =>   71.84%
      4 bugs found during  implementation   =>    3.88%
      2 bugs found during  design           =>    1.94%
      9 bugs found during  integration      =>    8.74%
      1 bugs found during  investigation    =>    0.97%
      1 bugs found during  functional test  =>    0.97%

When Fixed Statistics
  Of 80 bugs fixed:
     51 bugs fixed during  post-release     =>   63.75%
     13 bugs fixed during  alpha test       =>   16.25%
      2 bugs fixed during  beta test        =>    2.50%
      3 bugs fixed during  integration      =>    3.75%
      3 bugs fixed during  design           =>    3.75%
      6 bugs fixed during  investigation    =>    7.50%
      2 bugs fixed during  implementation   =>    2.50%

Severity Statistics
      Number of severity 1 bugs  =  10  =>   9.71%
      Number of severity 2 bugs  =  14  =>  13.59%
      Number of severity 3 bugs  =  65  =>  63.11%
      Number of severity 4 bugs  =  12  =>  11.65%
      Number of severity 5 bugs  =   2  =>   1.94%

Status Statistics
      Number of new bugs         =   1  =>   0.97%
      Number of open bugs        =   0  =>   0.00%
      Number of resolved bugs    =  84  =>  81.55%
      Number of postponed bugs   =   0  =>   0.00%
      Number of duplicate bugs   =   0  =>   0.00%
      Number of verified bugs    =   0  =>   0.00%
      Number of assigned bugs    =  18  =>  17.48%
      Number of integrated bugs  =   0  =>   0.00%
      Number of released bugs    =   0  =>   0.00%
```

Figure13-7 continued: DDTs Management Summary Report: General Statistics

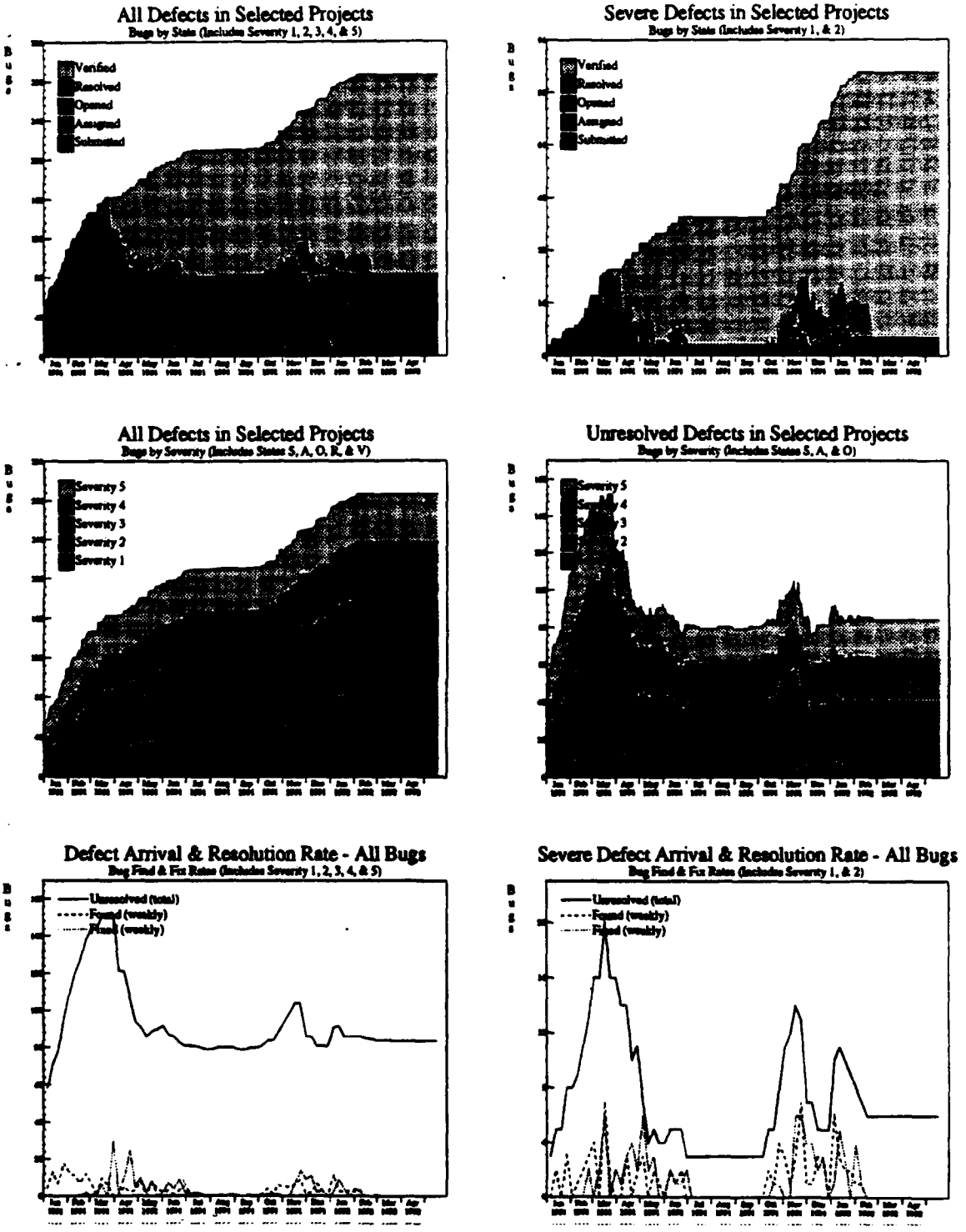


Figure 8. Examples of GUI Outputs

14. EXPERT DEBUGGING SOFTWARE ASSISTANT (EDSA)

EDSA is a browser that supports understanding and static analysis of Ada source code. It provides such capabilities as control and data flow browsing, pretty-printing, elision-based viewing, and search management. In addition, its annotation capability can support the conduct of code reviews and inspections, as well as capture the progress of formal verification activities.

14.1 Tool Overview

This product was developed by Array Systems Computing, Inc. and has been marketed since 1991. It has between 5 and 10 users. Array Systems Computing provides software consultancy and training, and performs independent verification and validation activities. Tool users are supported by a newsletter and hot-line support. EDSA is available under Unix, VMS, and DOS. An X-Windows version is available. The examination was performed on version 2.0 of this product running UNIX on a Sun-4 system with OpenWindows. At the time of evaluation, prices for EDSA started at $3,750.

Use of EDSA starts with parsing an Ada source code file. This produces an attributed syntax tree and symbol table that are stored in the user library. A successful parse is not required for browsing and any errors encountered during parsing are reported, together with appropriate warnings. When a program is contained in several files, compilation units must be parsed in compilation order. The output of the parser is used for browsing and EDSA can be invoked on any of the parsed files independently. If required, preparation of a pretty-printed output version of the original source file is available. This uses standardized indentation based on the parse tree to emphasize the control structures of the code.

EDSA supplements the traditional random traversal and string searching common to many browsers and editors with several logic-based traversal methods. These additional traversal methods allow a user to exploit the structure and meaning of the code. Specifically, the following types of traversal are supported:

- Random. Movement is achieved by usage of the cursor and scrolling keys, and by a set of defined *focus* commands.
- String searching. String commands find specified items in the code, for example, the statements that a particular statement depends upon.
- Syntax directed. This allows the user to follow paths defined by the syntax tree.

- Dependency. Provides for tracing back all the statements that the currently selected statement depends on (typically these statements are the controlling statement and the statements that define its input variable values).
- Data flow. Allows tracing back to where a variable was originally given a value and then ahead to every usage of that value until it is changed.
- Control flow. Follows the control logic of the source code.
- Object-usage. Allows visiting every statement where an Ada object (that is, a variable, parameter, component, or slice) appears in a specific context.

In each case, a stack and backtracking facilities are provided for switching between paths when more than one path can be followed.

When browsing, the source text is pretty-printed in the *view window*. This window changes as the file is a traversed or new views constructed. The *message window* displays the most recent commands entered and, sometimes, messages relating to the current command. The *response box* is a temporary window used to notify the user of problems with an entered command or to ask for verification of a command. During browsing, the user can switch to an editor and, at the end of the edit, cause the syntax tree and symbol table to be appropriately updated.

Views are provided to help mitigate the complexity of perusing large programs. Views are a selection of some or all of the statements in the source code. By showing only specific parts of the code, they allow a user to restrict himself to only those features of interest, for example, those portions of the code that are within a particular depth, that use a specified symbol, or that use a specified structure. Views can be created, modified, printed, and combined.

EDSA's statement annotations are useful for adding documentation to source code without modifying the original source file. Whereas comments exist in both the source code and syntax tree, annotations exist only in the syntax tree (although options to cause them to be included in the source code are provided). This can be useful for recording temporary observations during an analysis session or for recording other types of working notes. Depending on the value of a customization parameter, these annotations act like special comments or are hidden from view until required. After editing, EDSA can cause the syntax tree and symbol table to be appropriately updated and annotations inherited from the old tree, adjusted if necessary, to conform to the changes.

Pebbling is another type of annotation. Here the annotations, or *pebbles*, are used to record the fact that statements have been examined and that some conclusion about their

correctness has been reached. The pebbling feature uses dependency information to link each of a statement's inputs to all of the statements that might provide values to those inputs. It propagates correctness information by automating a generalization of the following rule from propositional logic: *Given that A is true, and that A being true implies that B is true, then it follows that B must be true. (A, $A \rightarrow B \Rightarrow B$).* The user places white pebbles to indicate that the statement and its contributors are assumed to be correct, that is, globally correct, for verification purposes. He places grey pebbles to indicate that only the statement itself is assumed to be correct, that is, locally correct. If the contributors to a locally correct statement are globally correct, EDSA automatically replaces a grey pebble with a black pebble to indicate that global correctness has been derived, although not asserted.

14.2 Observations

Ease of use. A user can interact with EDSA using a command line with auto-completion, cursor keys or a mouse to move around the menus and command line, or key bindings. In the latter case, default keys are bound to the most commonly used EDSA commands; the user can adjust these bindings to customize EDSA as desired. Additional opportunities for customization allow modifying text appearance and system parameters. Examples of system parameters include switches that specify whether annotations should be hidden and whether the user should be queried for backtracking to previously skipped paths. Expertise level is another system parameter. It allows a user to be assigned one of six levels of expertise that are used to determine the extent of help, menus, and warnings messages provided.

Documentation and user support. The documentation is extensive and includes several useful examples. Array Systems Computing were prompt and helpful in responding to queries.

Problems encountered. EDSA performed exactly as described in the documentation. No problems were encountered during its use.

14.3 Sample Outputs

Figures 14-1 through 14-6 provide sample outputs from EDSA.

```
separate ( Ll_Compile )
package body LL_TOKENS is
    ...

    procedure Advance( eos : out BOOLEAN; next : out LLTOKEN; more : in BOOLEAN
          :=TRUE ) is
        ...
        procedure Get_Char( char : out CHARACTER ) is
        begin
            if End_Of_File( Standard_Input ) then
                ...
            elsif End_Of_Line( Standard_Input ) then
                Skip_Line( Standard_Input );
                ...
            else
                Get( Standard_Input, char );
            end if;
            end Get_Char;
                                        ...
            end Next_String;
    begin
        ...
        -- Skip white space and comments
        while ( current_char=ASCII.ETX ) or ( current_char=ASCII.HT ) or (
              current_char=' ' ) or ( current_char='-' ) loop
            if current_char='-' then
                Look_Ahead;
                ...
                Skip_Line( Standard_Input );
                ...
            end if;
            Char_Advance;
        end loop;
        if current_char=ASCII.EOT then
            ...
        elsif current_char='"' then
            Next_String;
        elsif current_char=''' then
            Next_Character;
        elsif ( current_char in UPPER_CASE_LETTER ) or ( current_char in
              LOWER_CASE_LETTER ) then
            Next_Identifier;
        else
            Next_Spec_Sym;
        end if;
        ...
        end Advance;
end LL_TOKENS;
```

Figure 14-1. EDSA Threads View of Compilation Unit LL_TOKENS

```
separate ( Ll_Compile )
package body LL_TOKENS is
    ...

    procedure Advance( eos : out BOOLEAN; next : out LLTOKEN; more : in BOOLEAN
          :=TRUE ) is
        ...

        procedure Next_String is
            ...
        begin
            ...
            while current_char/='"' loop
                ...
                exit when End_Of_Line( Standard_Input );
                ...
            end loop;
            ...
        end Next_String;

    begin
        ...
        -- Skip white space and comments
        while ( current_char=ASCII.ETX ) or ( current_char=ASCII.HT ) or (
              current_char=' ' ) or ( current_char='-' ) loop
            if current_char='-' then
                ...
                exit when look_char/='-';
                ...
            end if;
            ...
        end loop;
        ...
    end Advance;
end LL_TOKENS;
```

Figure 14-2. EDSA Breaks View of Compilation Unit LL_TOKENS

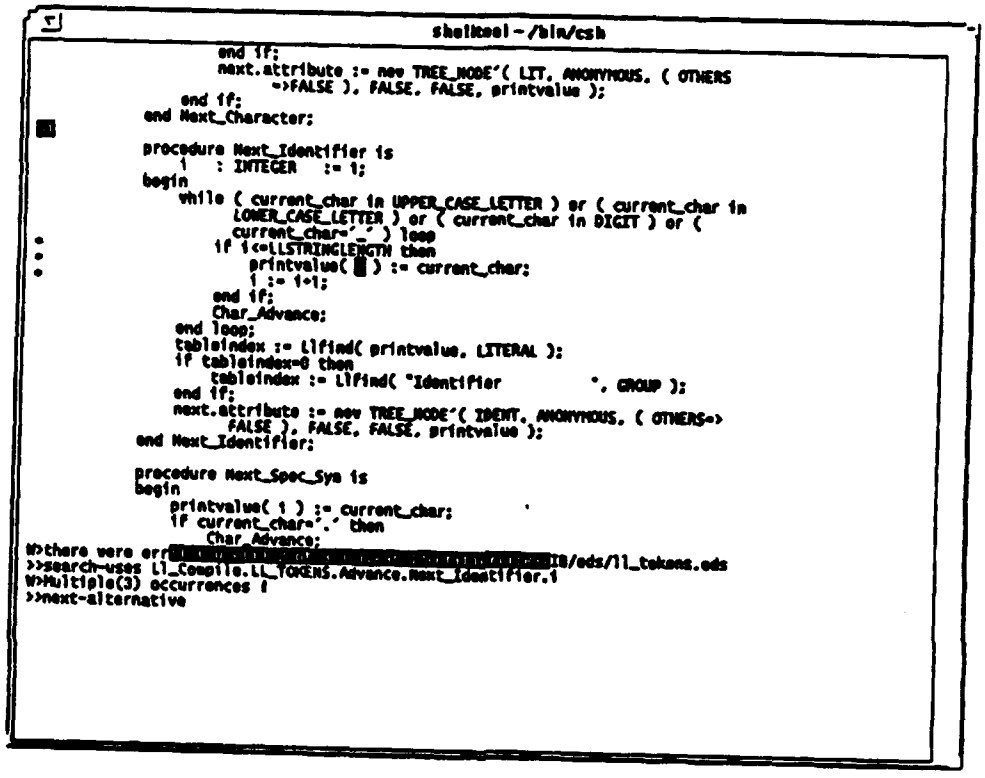


Figure 14-3. EDSA Screen of Statement Traversal Using Data Flow of Variable I

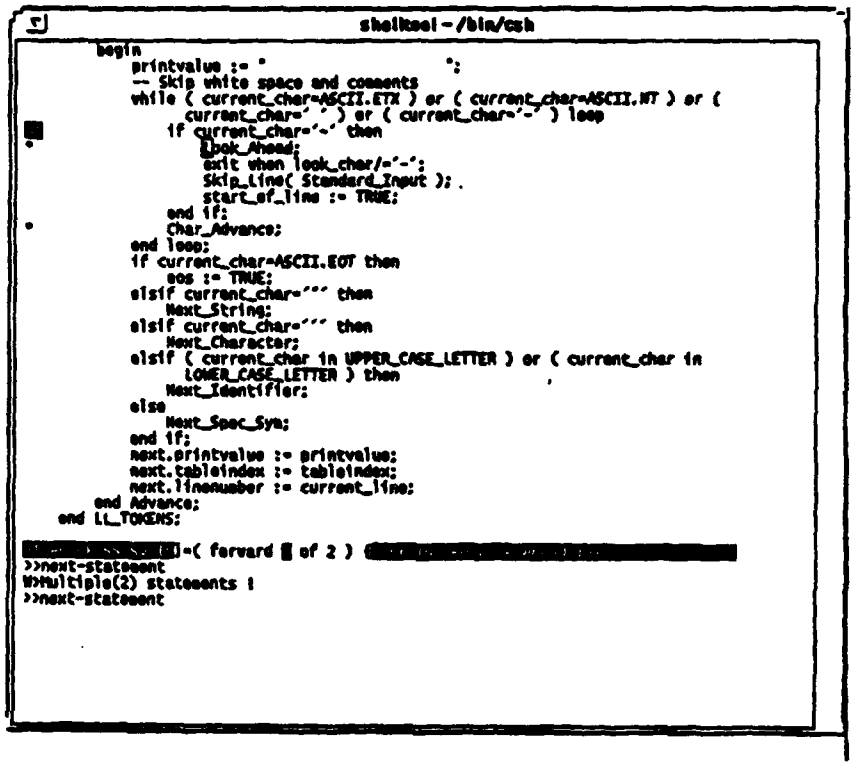


Figure 14-4. EDSA Screen of Statement Traversal Using Control Flow in Unit LL_TOKENS

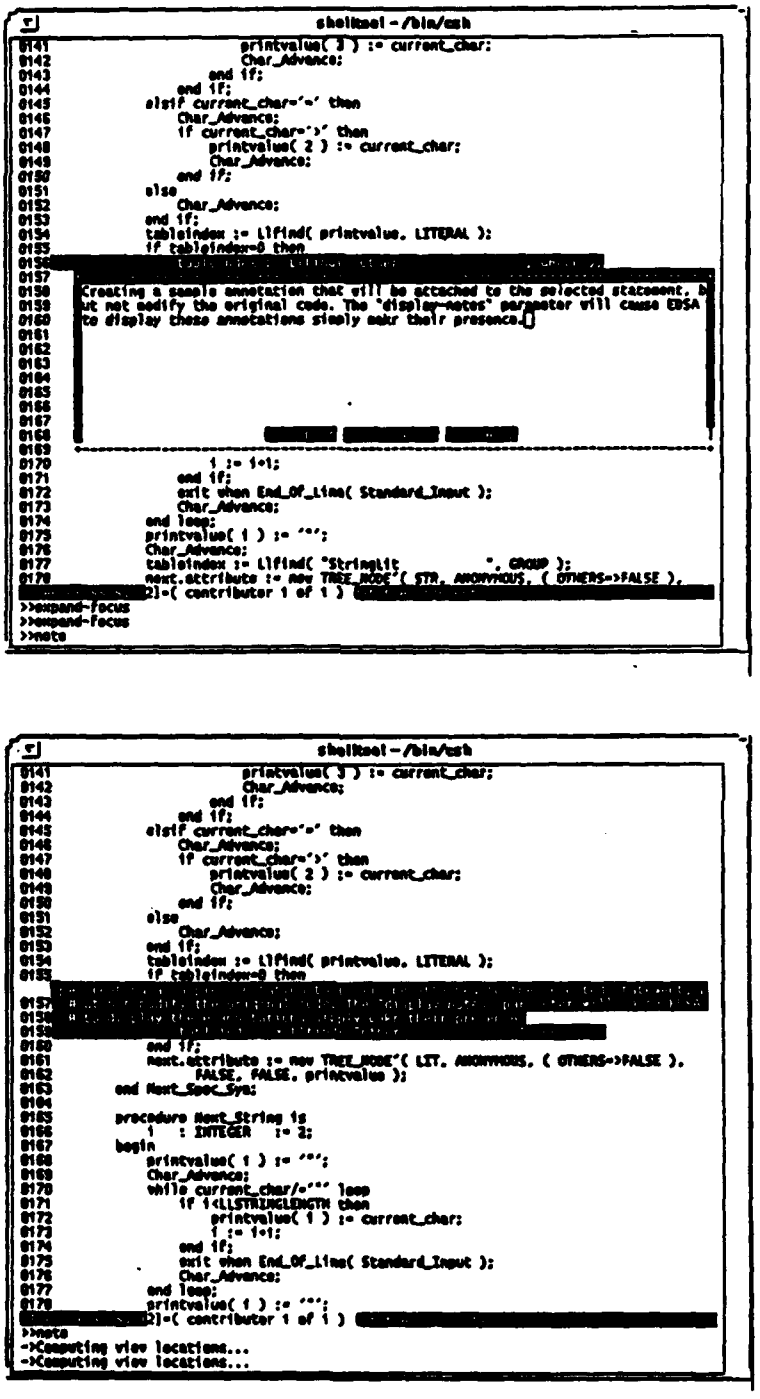


Figure 14-5. EDSA Annotations Example in Compilation Unit LL_TOKENS

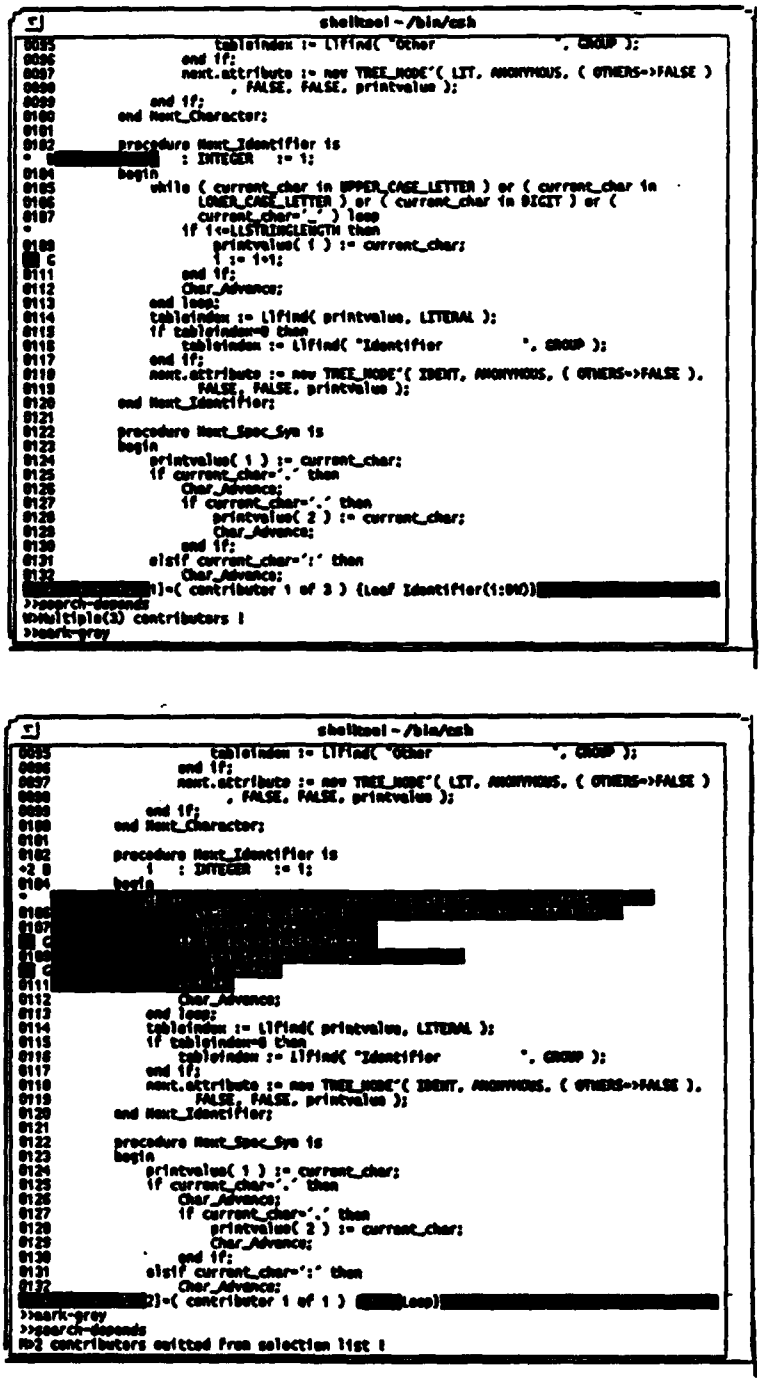


Figure 14-6. EDSA Pebbling Example in Compilation Unit LL_TOKENS

15. LDRA Testbed

The LDRA Testbed provides both static and dynamic analysis. In static analysis, source code is analyzed to give information on control, data, and information flow, logical complexity, and procedure and variable usage. Conformance to user-weighted programming standards is checked. Dynamic analysis capabilities provide assertion analysis and measurement of test completeness in terms of subcondition, statement, branch, and Linear Code Sequence and Jump (LCSAJ) coverage. Analysis of test data set redundancy is provided to optimize the test effort. Identification of the test data sets that execute each line of code facilitates software modification.

15.1 Tool Overview

The LDRA Testbed was developed by Liverpool Data Research Associates. It is marketed by a subsidiary company, Program Analysers Ltd., who also provide a series of training courses and consultancy services. Additional third party products are available in Europe. The Testbed has been commercially available since 1974 and there are over 400 current licensees. It is available for eight languages (Ada, C, Fortran, Pascal, PL/M 86, PL/1, COBOL, and Coral 66) on a wide range of operating environments. The following partial list exemplifies the scope of this range of environments: Apollo machines under Unix, DEC VAX under VMS, Unix, or Ultrix, IBM under CMS, DOS or TSO/MVS, Sun 3 and Sun 4 under Unix, and Hewlett-Packard under RTEA or HPUX. Using windowing capabilities, a graphical interface is available for Sun, Apollo and, in some cases, VAX workstations. The command line options available for the VAX/VMS and Unix environments differ. Unix users can set options to expand included files where possible, generate diagnostic printouts, and initialize test profiles. Several additional options are provided in the VAX/VMS environment, for example, the ability to create a log file of LDRA Testbed usage, to limit the type of coverage monitored, and to format or pack the generated execution history.

The evaluation was performed on version 4.8.01 of the Ada testbed running on a Sun 4 under Unix. At the time of evaluation, prices for the testbed start at $12,000, depending on the class of computer and language.

LDRA Testbed is a menu-driven tool. Its application begins with static analysis of the software under test. This lexical and syntactic analysis produces a reference listing containing source code reformatted to LDRA Testbed reformatting standards. Reference line num-

bers are given for each statement line. At the same time, the source code is searched for violations to the applicable set of language standards provided with the testbed. These standards check for conformance to much of the Safe Ada Subset. Reporting on any particular standard is optional, the user selects appropriate standards by awarding penalty marks greater than zero for violations; the static analysis produces a total penalty award for the analyzed source code. Where appropriate, the user can also specify acceptable limits for particular standards.

Complexity analysis is based on the control flow structure expressed in terms of basic blocks. The complexity is reported in terms of the number and average length of basic blocks, the number of control flow knots, and cyclomatic complexity. In addition, two approaches are used to analyze program structure. First, interval analysis reports on the reducibility of the software and degree of nesting. Second, the program structure is evaluated against a set of user-tailorable language construct templates, an approach called structured programming verification. Two further metrics, essential knots and McCabe's essential complexity, are provided to report on unstructuredness. The user specifies whether complexity analysis should be applied to all program units or limited to an identified set of program units. Kiviat diagrams are provided for reporting of the various complexity and structure metrics. These diagrams allow diagramming multiple metrics simultaneously, each with its achieved and user-defined upper and lower bounds.

The Data Flow Analyser reports procedure call information, data flow anomalies, and procedure parameter analysis. Weak data flow analysis is applied to identify undefined data variables and defined variables that are redefined or undefined without first referencing the previous definition. Procedure parameter analysis classifies parameters as referenced only, defined only, both referenced and defined, or not used; this analysis is carried out across procedure boundaries.

Information flow analysis is a new capability that provides information on the interdependencies of program variables. LDRA Testbed currently supports analysis of backwards strong and weak dependencies on a procedure-by-procedure basis. This capability can be used in two ways. First, as a source of documentation, for example, to support identifying the consequences of a software change. Secondly, the user can specify information flow dependency assertions as special comments. The testbed then compares the expected dependencies with actual dependencies, reporting the results.

The Cross Referencer performs a complete cross-reference of all data items used in a program. The type of each data item is classified as global, local, or parameter. For each

procedure, the referencer also identifies all other procedures that this procedure calls, and all procedures that call this one.

LCSAJ analysis is the final type of static analysis provided. It aids the user in isolating LCSAJs by highlighting, on a source code listing, the start and finish of the linear code sequence of each LCSAJ. Unreachable LCSAJs, and any other unreachable code statements, are indicated.

The Dynamic Analyser instruments source code with probes which, upon execution, write information to an execution history file. This is usually done by writing to the host disk at run time. To allow for host/target computer configurations, however, the instrumentation can be adapted to channel the execution history generated by the instrumented target image back to the host and stored for subsequent analysis. This may be achieved by using a spare serial line. Alternatively, it may be possible to arrange for storage of the execution history using an area of memory on the target, with this buffer subsequently uploaded to the host.

After instrumentation, the user compiles and links the instrumented program in the usual way. For simple programs, the resulting executable can be run under control of LDRA Testbed, which queries the user for the names of input and output streams. Alternatively, the program can be executed independently of the testbed. In either case, after the program has run, the user invokes the Dynamic Coverage Analyser to analyze the generated execution history and provide a name for the current test data set. The coverage analyzer takes account of the results of previous test data sets to accumulate the execution coverage over a series of test runs. (The user has no direct control over adding an execution history to the accumulated coverage data; this is handled automatically.) For each of subcondition, statement, branch, and LCSAJ coverage, the analyzer provides a list of the respective items contained in the program and identifies the old, new, and total coverage percentage achieved for each item. Unexecuted items are identified. In each case, this is followed by a summary that reports the total number of executable statements, branches, or LCSAJs, as appropriate, the number that were executed, the number not executed, and the corresponding test effectiveness metric.

The user may request a dynamic trace to explicitly show the flow of control resulting from the test data set. This trace may be limited to specified procedures, or to between a user-specified range of code line numbers. The LDRA Testbed will override this request if the resulting display will be too large.

The testbed uses three Test Effectiveness Ratio (TER) metrics to report on the effectiveness of the test data:

- TER1 = Number of statements exercised at least once / Total number of statements
- TER2 = Number of branches exercised at least once / Total number of branches
- TER3 = Number of LCSAJs exercised at least once / Total number of LCSAJs

In terms of coverage, TER3 lies between branch and path coverage. That is, LCSAJs provide a measure that is more stringent than branch coverage without incurring the overhead of path coverage. Additionally, LDRA Testbed reports the number of overlapping LCSAJs containing each reformatted statement as the "density." This figure can be used as a measure of the complexity encountered when reading or modifying the program.

When a program contains tasks, the generated execution history will contain the interleaved execution histories of those tasks. LDRA Testbed can distinguish between these multiple histories, but some special user actions are required to assist in the processing of the separated histories.

The user can embed assertions in Ada comments. These special comments can be used to specify pre- or post-conditions applying to a section of code, check that inputs satisfy predetermined ranges, or check that loop and array indices are within bounds. When conformance checking is switched on, the testbed translates the special comments to executable code and inserts a user-tailorable failure handling routine. The supplied failure handling routine simply prints a message identifying the failing assertion and then raises a fail exception. It is the user's responsibility to determine appropriate assertion conditions and to ensure that the assertions are positioned where valid executable statements are allowed in the source code. The assertion format and, to some extent, syntax and semantics are tailorable via means of a parameter file.

Two final capabilities provide some limited support for regression analysis. A Profile Analyzer is provided to compare the coverage profiles generated by a series of test data sets. It identifies any data set(s) that are redundant, that is, those that do not contribute to increasing the overall coverage. Where two or more redundant test data sets are identified, LDRA Testbed will recommend removal of the one that generates the largest execution history. For each executable line of code, the Dynamic Data Set Analysis option identifies the test data set(s) that execute that line. This allows the user to determine which test data sets are affected by a modification and, therefore, the tests that must be repeated.

The results of testbed analysis are examined using a viewing option. They can be viewed at either the compilation unit level or system level (that is, for the full set of compilation units). Various textual displays are available or the user may access a submenu of graphical displays. Navigation through textual displays is command driven. Graphical displays are available as bar charts of complexity and coverage measures, Kiviat diagrams of quality metrics, flowgraphs of the software control flow graph, and call-trees showing the procedure hierarchy. Static and dynamic views of both call-trees and flowgraphs are available. The static control flow graph can be annotated with the results of coverage analysis. In addition, the user can request an active flowgraph that illustrates the execution achieved by the last test data set. Navigation through the graphical displays is provided via selection from a set of icons that support such functions as automatic zooming and printing a screen.

15.2 Observations

Ease of use. Overall, LDRA Testbed is very easy to use and provides a broad range of testing facilities. It automates all repetitive tasks and requires no redundant user input; for example, a special script is provided to facilitate testing of software composed of many source modules in separate files. This script, called *tbset*, allows a user to associate a name with a set of files and manipulate, list, and select sets. LDRA Testbed can be invoked from *tbset* to apply user-selected testbed operations to a chosen set of files as a group. In this mode, however, some usual testbed options are not available; in particular, the user cannot limit processing to a named set of files or limit reporting to a named set of program units. In addition, only system-wide analysis results, a call-tree display, and a variety of flow graphs are available for viewing. (To access results for particular software units, the user can view results through the testbed directly.) If necessary, *tbset* allows the user to spawn a shell script for non-testbed related processing.

The Management Summary report provides useful high-level information on the quality of the software and on the effectiveness of testing to date. More detailed information is provided in a series of analysis reports, some of which are very lengthy. While some users may find the provision of multiple alternative complexity measures useful, others will find much of this information redundant.

Graphical outputs are available on Sun, Apollo, HP, IBM RS6000, VAX, and most other types of workstations with windowing capabilities, histograms drawn in orthographic projection are available for any terminal supporting VT100 graphics. These histograms are

used to profile coverage information and summarize information on the program quality, complexity and structuredness. Full color is available for graphical displays. All graphics displays can be exported as Postscript files.

Documentation and user support. The supplied documentation is well-written and comprehensive. It includes a standard interface file to facilitate using LDRA Testbed outputs as inputs to other tools. This file allows testbed information to be viewed at three levels: procedure/function, source module, and project (that is, some related set of source modules). Through this interfacing facility, LDRA Testbed has been used with StP, Teamwork, Typhoon, System Engineer, Mascot, Infomix, and ASA CASE tools, and with the TBGEN testing tool.

In all instances, the staff at Program Analysers were helpful and friendly and provided quick resolution of encountered problems.

Instrumentation overhead. Full instrumentation of the Ada Lexical Generator for statement, branch, and LCSAJ coverage gave a source code increase of nearly 100%. The size of the instrumented executable program increased approximately 12%. The user can limit the amount of instrumentation performed by requesting monitoring of only statement coverage, or only statement and branch coverage. Since the user specifies the files which are to be instrumented, instrumentation can be restricted to specific compilation units. It cannot, however, be limited to specific program units within a compilation unit.

Ada restrictions. The LDRA Testbed supports full Ada. However, the documentation lists the following constraints for version 4.8.01 of the Testbed:

- For static analysis and cross-referencing: (1) The use of generics may not be correctly handled in some cases, (2) Analyses are limited to variables in the current module, (3) Overloaded procedures may cause misleading messages about recursion, and (4) Some combinations of literal procedure parameters are incorrectly analyzed.
- Calls between procedures in package bodies are only handled if their declaratives appear textually before use.
- Incorrect branches are generated for certain nested select statements.
- In the case of information flow analysis, strongly-defined variables may be miscategorized in the presence of exception handlers.
- The analysis may be incorrect for loops implementing recursive functions of degree greater than two.

Problems encountered. Difficulties encountered in installing an earlier version of the testbed have been resolved. LDRA Testbed performed as described in the documentation.

15.3 Planned Additions

Currently under beta testing, dynamic data flow testing for Ada is expected to become available in autumn 1992. Also under development are system-wide data flow analysis and the assessment and reporting of reliability metrics.

15.4 Sample Outputs

Figures 15-1 through 15-19 provide sample outputs from LDRA Testbed.

```
**********************************************************************
**********                                                  **********
**********              MANAGEMENT SUMMARY                  **********
**********                                                  **********
**********************************************************************

TESTBED VERSION   : 4.8.01
FILE UNDER TEST   : adalex_dir_2/ll_compile.a
DATE OF ANALYSIS  : Mon Oct 12 11:53:18 EDT 1992

STANDARDS VIOLATIONS IN STATIC ANALYSIS
---------------------------------------

=============================================================================
 LINE                                                              PENALTY
NUMBER                        VIOLATION                           MARK
=============================================================================
   28          I-O package                                          1
   82          USE clause                                           1
   82          I-O package                                          1
   85          Exception declaration                                1
   88          Number Declaration                                   1
   95          Predefined language environment name "INTEGER"       1
                                  ...
  688          Identical name in scope "TESTSYNCH"                  1
  694          Predefined language environment name "INTEGER"       1
  695          Identical name in another scope "I"                  1
  695          Predefined language environment name "INTEGER"       1
  744          Predefined language environment name "FALSE"         1
  752          Raise statement                                      1
  782          Predefined language environment name "TRUE"          1
  797          Predefined language environment name "TRUE"          1

-----------------------------------------------------------------------------
TOTAL PENALTY FROM STATIC ANALYSIS = 125
TOTAL NUMBER OF LINES IN PROGRAM   = 868

SUMMARY OF EXECUTABLE BODIES
----------------------------

=================================
                      START   NO OF
NAME                  LINE    LINES
=================================
LLFIND                152      22
LLPRTSTRING           179      10
LLPRTTOKEN            194      11
LLSKIPTOKEN           210      10
LLSKIPNODE            225      12
LLSKIPBOTH            244      13
LLFATAL               262       9
GET_CHARACTER         278      14
CVT_STRING            307      10
MAKE_TOKEN            318      66
```

Figure 15-1. LDRA Testbed Management Summary for LL_COMPILE

```
LLNEXTTOKEN             400       8
BUILDRIGHT              455      66
BUILDSELECT             527       8
READGRAM                537      39
ERASE                   587      14
MATCH                   620      17
EXPAND                  639      47
SYNCHRONIZE             697      62
TESTSYNCH               761      15
PARSE                   778      75
LLMAIN                  855       7
LL_COMPILE              864       5
---------------------------------

          THERE ARE    1 UNREACHABLE LCSAJS
          THE MAXIMUM LCSAJ DENSITY IS   16 AT LINE  458

          THERE ARE    1 SEQUENCES OF UNREACHABLE CODE
          THE LONGEST IS    2 LINES AT LINE  167
          THE TOTAL NUMBER OF UNREACHABLE LINES IS   10

          THERE ARE    7 UNREACHABLE BRANCHES

1COMPLEXITY ANALYSIS PRODUCES THE FOLLOWING TABLE OF RESULTS
-----------------------------------------------------------
```

PROCEDURE	EXEC. LINES	BASIC BLOCKS	AVG. LEN.	ORDER 1 INTERV.	MAX ORDER INTERV.	REDUC.	MCCABE	KNOTS	MCCABE	E KNOTS
LL_COMPILE	27	1	27.00	1	1	YES	1	0	1	0
LLFIND	21	13	1.62	2	2	YES	5	14	4	6
LLPRTSTRING	10	5	2.00	2	2	YES	3	2	3	2
LLPRTTOKEN	10	4	2.50	1	1	YES	2	1	1	0
LLSKIPTOKEN	10	1	10.00	1	1	YES	1	0	1	0
LLSKIPNODE	12	1	12.00	1	1	YES	1	0	1	0
LLSKIPBOTH	13	1	13.00	1	1	YES	1	0	1	0
LLFATAL	9	2	4.50	1	1	YES	1	0	1	0
GET_CHARACTER	14	6	2.33	1	1	YES	3	2	1	0
MAKE_TOKEN	68	17	4.00	1	1	YES	11	20	1	0
LLNEXTTOKEN	8	3	2.67	1	1	YES	2	0	1	0
LLMAIN	14	1	14.00	1	1	YES	1	0	1	0
CVT_STRING	10	7	1.43	2	2	YES	3	2	1	0
READGRAM	38	14	2.71	5	3	YES	6	5	1	0
PARSE	73	23	3.17	2	2	YES	11	11	1	0
BUILDRIGHT	62	20	3.10	2	2	YES	10	30	8	26
BUILDSELECT	8	4	2.00	2	2	YES	2	1	1	0
ERASE	11	6	1.83	2	2	YES	3	4	3	4
EXPAND	43	15	2.87	4	2	YES	7	2	1	0
TESTSYNCH	14	7	2.00	2	2	YES	3	2	1	0

Figure 15-1 continued: LDRA Testbed Management Summary for LL_COMPILE

MATCH	16	9	1.78	2	2	YES	4	9	4	
SYNCHRONIZE	59	23	2.57	7	3	YES	9	11	4	
TOTAL	550	183	3.01	23	3	YES	69	116	21	5

THE PROGRAM CONTAINS 22 PROCEDURES

1STANDARDS VIOLATIONS IN COMPLEXITY ANALYSIS

PROCEDURE	VIOLATION	PENALTY
LLFIND	CONTAINS ESSENTIAL KNOTS	1
LLPRTSTRING	CONTAINS ESSENTIAL KNOTS	1
MAKE_TOKEN	MCCABE MEASURE GREATER THAN 10	1
PARSE	MCCABE MEASURE GREATER THAN 10	1
BUILDRIGHT	CONTAINS ESSENTIAL KNOTS	1
ERASE	CONTAINS ESSENTIAL KNOTS	1
MATCH	CONTAINS ESSENTIAL KNOTS	1
SYNCHRONIZE	CONTAINS ESSENTIAL KNOTS	1

TOTAL PENALTY FROM COMPLEXITY ANALYSIS = 8

1DATA FLOW ANALYSIS RESULTS

1 VARIABLES WERE DECLARED BUT NEVER USED
40 TYPE UR ANOMALIES FOUND
17 TYPE DU ANOMALIES FOUND
41 TYPE DD ANOMALIES FOUND

1DYNAMIC COVERAGE ANALYSIS REPORT

PROFILES INCLUDED FOR THE FOLLOWING TEST DATA SETS

1) test1.lex
2) sample.lex

DYNAMIC ANALYSIS WARNINGS

8 MISSING LINEAR CODE SEQUENCE AND JUMP TRIPLES
6 MISSING BRANCHES

1STATEMENT EXECUTION HISTORY SUMMARY

	EXECUTABLE STATEMENTS	NUMBER EXECUTED			TER 1		
		OLD	NEW	TOTAL	OLD	NEW	TOTAL
LL_COMPILE	27	27	27	27	1.00	1.00	1.00
LLFIND	17	16	17	17	0.94	1.00	1.00
LLPRTSTRING	10	0	0	0	0.00	0.00	0.00
LLPRTTOKEN	10	0	0	0	0.00	0.00	0.00
LLSKIPTOKEN	10	0	0	0	0.00	0.00	0.00

Figure 15-1 continued: LDRA Testbed Management Summary for LL_COMPILE

LLSKIPNODE	12	0	0	0	0.00	0.00	0.00
LLSKIPBOTH	13	0	0	0	0.00	0.00	0.00
LLFATAL	8	0	0	0	0.00	0.00	0.00
GET_CHARACTER	14	0	0	0	0.00	0.00	0.00
MAKE_TOKEN	67	0	0	0	0.00	0.00	0.00
LLNEXTTOKEN	8	8	8	8	1.00	1.00	1.00
LLMAIN	14	14	14	14	1.00	1.00	1.00
CVT_STRING	9	0	0	0	0.00	0.00	0.00
READGRAM	38	38	38	38	1.00	1.00	1.00
PARSE	73	56	56	56	0.77	0.77	0.77
BUILDRIGHT	62	54	54	54	0.87	0.87	0.87
BUILDSELECT	8	8	8	8	1.00	1.00	1.00
ERASE	11	11	11	11	1.00	1.00	1.00
EXPAND	43	38	38	38	0.88	0.88	0.88
TESTSYNCH	14	0	0	0	0.00	0.00	0.00
MATCH	15	13	13	13	0.87	0.87	0.87
SYNCHRONIZE	57	0	0	0	0.00	0.00	0.00
TOTAL	540	283	284	284	0.52	0.53	0.53

SUB-CONDITIONS SUMMARY

SUB-CONDITIONS		NUMBER EXECUTED OLD	NEW	TOTAL	TER CON OLD	NEW	TOTAL
LL_COMPILE	PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLFIND	PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLPRTSTRING	PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLPRTTOKEN	PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLSKIPTOKEN	PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLSKIPNODE	PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLSKIPBOTH	PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLFATAL	PROCEDURE CONTAINS NO SUB-CONDITIONS						
GET_CHARACTER	PROCEDURE CONTAINS NO SUB-CONDITIONS						
MAKE_TOKEN	PROCEDURE CONTAINS NO SUB-CONDITIONS						
	...						
BUILDRIGHT	PROCEDURE CONTAINS NO SUB-CONDITIONS						
BUILDSELECT	PROCEDURE CONTAINS NO SUB-CONDITIONS						
ERASE	PROCEDURE CONTAINS NO SUB-CONDITIONS						
EXPAND	8	8	8	8	1.00	1.00	1.00
TESTSYNCH	PROCEDURE CONTAINS NO SUB-CONDITIONS						
MATCH	4	4	4	4	1.00	1.00	1.00
SYNCHRONIZE	12	0	0	0	0.00	0.00	0.00
TOTAL	24	12	12	12	0.50	0.50	0.50

Figure 15-1 continued: LDRA Testbed Management Summary for LL_COMPILE

1BRANCH EXECUTION HISTORY SUMMARY

		NUMBER EXECUTED			TER 2		
	BRANCHES	OLD	NEW	TOTAL	OLD	NEW	TOTAL
LL_COMPILE	13	13	13	13	1.00	1.00	1.00
LLFIND	34	11	12	12	0.32	0.35	0.35
LLPRTSTRING	9	0	0	0	0.00	0.00	0.00
LLPRTTOKEN	9	0	0	0	0.00	0.00	0.00
LLSKIPTOKEN	2	0	0	0	0.00	0.00	0.00
LLSKIPNODE	3	0	0	0	0.00	0.00	0.00
LLSKIPBOTH	3	0	0	0	0.00	0.00	0.00
LLFATAL	2	0	0	0	0.00	0.00	0.00
GET_CHARACTER	7	0	0	0	0.00	0.00	0.00
MAKE_TOKEN	31	0	0	0	0.00	0.00	0.00
LLNEXTTOKEN	5	4	4	4	0.80	0.80	0.80
LLMAIN	5	5	5	5	1.00	1.00	1.00
CVT_STRING	7	0	0	0	0.00	0.00	0.00
READGRAM	20	20	20	20	1.00	1.00	1.00
PARSE	39	25	25	25	0.64	0.64	0.64
BUILDRIGHT	26	22	22	22	0.85	0.85	0.85
BUILDSELECT	4	4	4	4	1.00	1.00	1.00
ERASE	7	7	7	7	1.00	1.00	1.00
EXPAND	18	15	15	15	0.83	0.83	0.83
TESTSYNCH	11	0	0	0	0.00	0.00	0.00
MATCH	11	7	7	7	0.64	0.64	0.64
SYNCHRONIZE	26	0	0	0	0.00	0.00	0.00
TOTAL	292	133	134	134	0.46	0.46	0.46

1LCSAJ EXECUTION HISTORY SUMMARY

		NUMBER EXECUTED			TER 3		
	LCSAJS	OLD	NEW	TOTAL	OLD	NEW	TOTAL
LL_COMPILE	13	13	13	13	1.00	1.00	1.00
LLFIND	34	10	11	11	0.29	0.32	0.32
LLPRTSTRING	10	0	0	0	0.00	0.00	0.00
LLPRTTOKEN	13	0	0	0	0.00	0.00	0.00
LLSKIPTOKEN	2	0	0	0	0.00	0.00	0.00

Figure 15-1 continued: LDRA Testbed Management Summary for LL_COMPILE

LLSKIPNODE	3	0	0	0	0.00	0.00	0.00
LLSKIPBOTH	3	0	0	0	0.00	0.00	0.00
LLFATAL	2	0	0	0	0.00	0.00	0.00
GET_CHARACTER	6	0	0	0	0.00	0.00	0.00
MAKE_TOKEN	33	0	0	0	0.00	0.00	0.00
LLNEXTTOKEN	7	3	3	3	0.43	0.43	0.43
LLMAIN	5	5	5	5	1.00	1.00	1.00
CVT_STRING	9	0	0	0	0.00	0.00	0.00
READGRAM	25	20	20	20	0.80	0.80	0.80
PARSE	34	23	23	23	0.68	0.68	0.68
BUILDRIGHT	32	24	24	24	0.75	0.75	0.75
BUILDSELECT	5	4	4	4	0.80	0.80	0.80
ERASE	8	7	7	7	0.88	0.88	0.88
EXPAND	20	15	15	15	0.75	0.75	0.75
TESTSYNCH	12	0	0	0	0.00	0.00	0.00
MATCH	12	6	6	6	0.50	0.50	0.50
SYNCHRONIZE	38	0	0	0	0.00	0.00	0.00
TOTAL	326	130	131	131	0.40	0.40	0.40

1SUMMARY OF EFFECT OF CURRENT TEST DATA SET ON THE COVERAGE METRICS

PROCEDURE NAME	TER 1	TER 2	TER 3
LL_COMPILE	1.00	1.00	1.00
LLFIND	Increased	Increased	Increased
LLPRTSTRING	No Change	No Change	No Change
LLPRTTOKEN	No Change	No Change	No Change
LLSKIPTOKEN	No Change	No Change	No Change
LLSKIPNODE	No Change	No Change	No Change
LLSKIPBOTH	No Change	No Change	No Change
LLFATAL	No Change	No Change	No Change
GET_CHARACTER	No Change	No Change	No Change
MAKE_TOKEN	No Change	No Change	No Change
LLNEXTTOKEN	1.00	No Change	No Change
LLMAIN	1.00	1.00	1.00
CVT_STRING	No Change	No Change	No Change
READGRAM	1.00	1.00	No Change
PARSE	No Change	No Change	No Change
BUILDRIGHT	No Change	No Change	No Change
BUILDSELECT	1.00	1.00	No Change
ERASE	1.00	1.00	No Change
EXPAND	No Change	No Change	No Change
TESTSYNCH	No Change	No Change	No Change
MATCH	No Change	No Change	No Change
SYNCHRONIZE	No Change	No Change	No Change

Figure 15-1 continued: LDRA Testbed Management Summary for LL_COMPILE

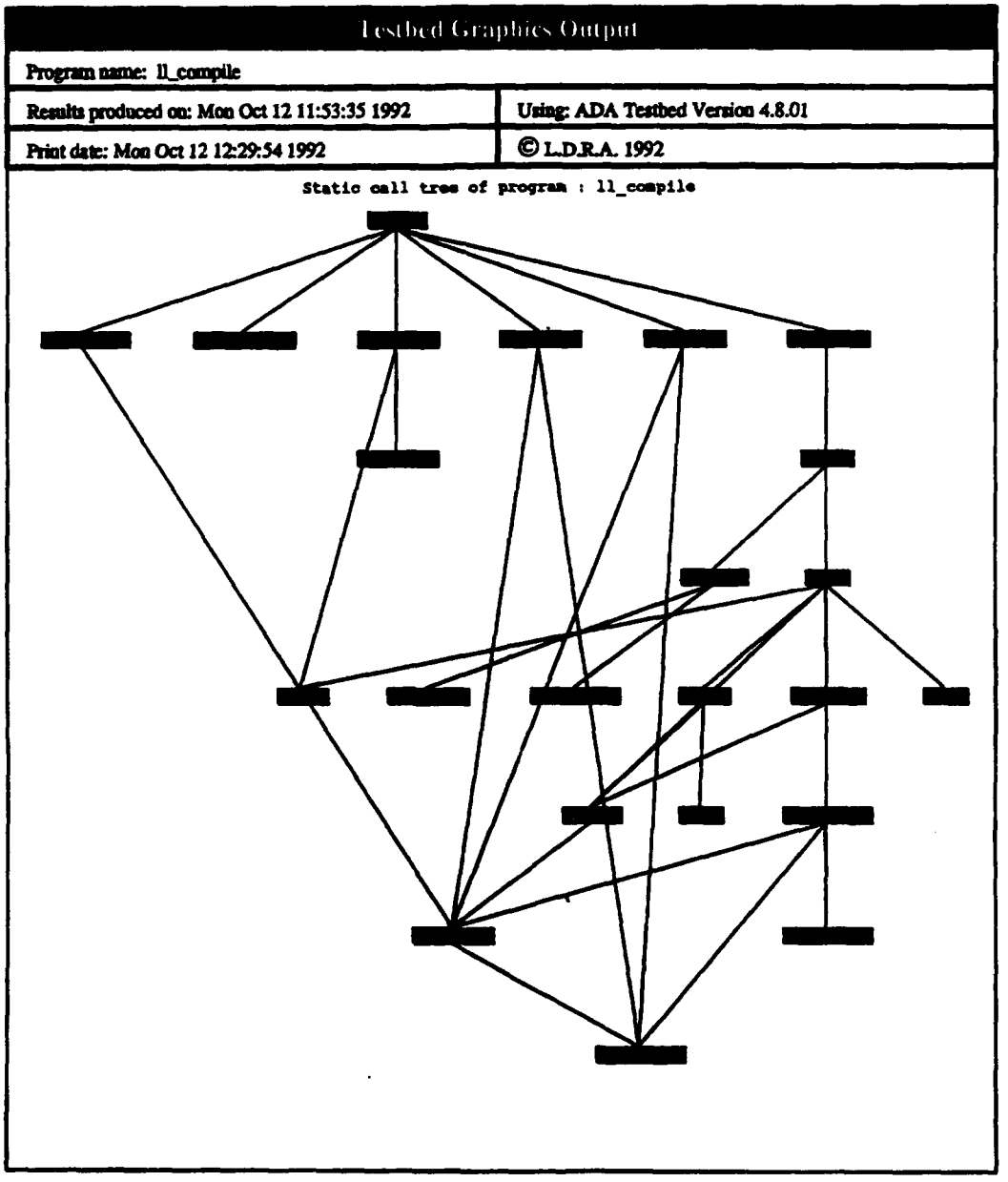


Figure 15-2. LDRA Testbed Static Call Tree of LL_COMPILE

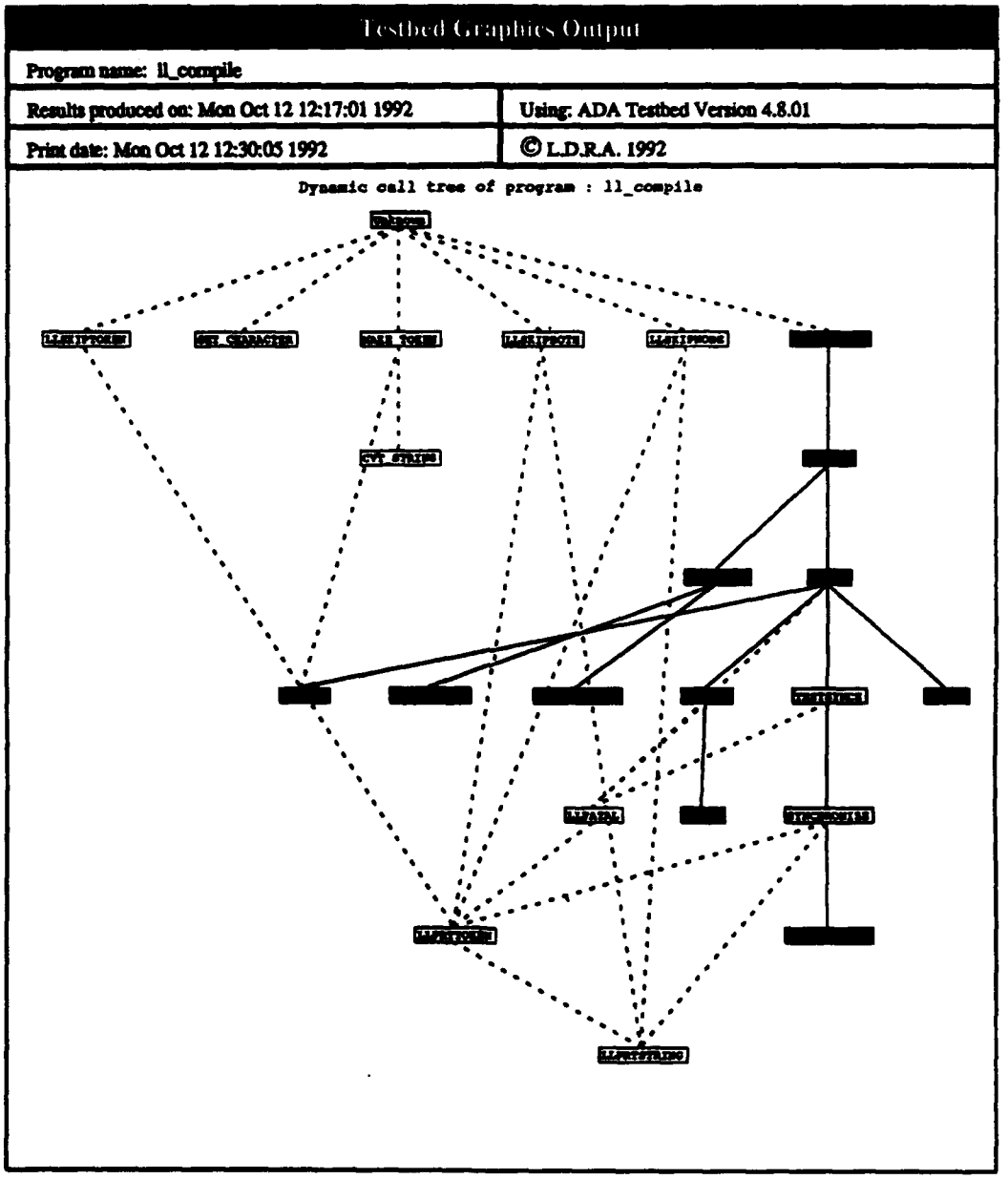


Figure 15-3. LDRA Testbed Dynamic Call Tree of LL_COMPILE

```
--------------------------
PROCEDURE CALL INFORMATION
--------------------------

-----------------------------

THE MAIN PROGRAM
BETWEEN LINES      31 AND     868
LL_COMPILE

CALLS THE FOLLOWING PROCEDURES
LLMAIN

-----------------------------

PROCEDURE
LLFIND
BETWEEN LINES     146 AND     174

DOES NOT CALL ANY INTERNAL PROCEDURES

IS CALLED BY THE FOLLOWING PROCEDURES
MAKE_TOKEN
PARSE

-----------------------------

PROCEDURE
LLPRTSTRING
BETWEEN LINES     176 AND     188

DOES NOT CALL ANY INTERNAL PROCEDURES

IS CALLED BY THE FOLLOWING PROCEDURES
LLPRTTOKEN
LLSKIPNODE
LLSKIPBOTH
SYNCHRONIZE

-----------------------------
                                    ...
1-------------------------------------------------
THE FOLLOWING VARIABLES WERE DECLARED BUT NEVER USED
VARIABLE            DECLARED ON LINE
-------------------------------------------------
TABLEINDEX                  453
```

Figure 15-4. LDRA Testbed Data Flow Analysis of LL_COMPILE

```
1================================================
                TYPE UR ANOMALIES
VARIABLE     UNDEFINE                          REFERENCE
================================================
'GLOBAL'STANDARD_ERROR
                180                              180
'GLOBAL'STANDARD_ERROR
                200                              200
                                  ...
RHSARRAY        436                              859 IN PROCEDURE PARSE
'GLOBAL'LLTABLESIZE
                136                              136
LLSYMBOLTABLE
                136                              866 IN PROCEDURE LLMAIN
LLEOTOKS        132                              866 IN PROCEDURE LLMAIN
LLCURTOK.PRINTVALUE
                135                              866 IN PROCEDURE LLMAIN
LLCURTOK.TABLEINDEX
                135                              866 IN PROCEDURE LLMAIN
LLCURTOK.LINENUMBER
                135                              866 IN PROCEDURE LLMAIN
LLCURTOK.ATTRIBUTE
                135                              866 IN PROCEDURE LLMAIN

1================================================
                TYPE DU ANOMALIES
VARIABLE     DEFINE                            UNDEFINE
================================================
CHILDCOUNT      457                              520
I               529                              532
LOCOFANY        788                              852
PRODUCTIONS     857 IN PROCEDURE READGRAM        861
RHSARRAY        857 IN PROCEDURE READGRAM        861
THISRHS         857 IN PROCEDURE READGRAM        861
LLSTACK.DATA    866 IN PROCEDURE LLMAIN          868
LLSTACK.ATTRIBUTE
                866 IN PROCEDURE LLMAIN          868
LLSTACK.PARENT
                866 IN PROCEDURE LLMAIN          868
LLSTACK.TOP     866 IN PROCEDURE LLMAIN          868
LLSTACK.LASTCHILD
                866 IN PROCEDURE LLMAIN          868
LLSYMBOLTABLE
                866 IN PROCEDURE LLMAIN          868
LLCURTOK.TABLEINDEX
                866 IN PROCEDURE LLMAIN          868
LLCURTOK.PRINTVALUE
                866 IN PROCEDURE LLMAIN          868
LLSENTPTR       866 IN PROCEDURE LLMAIN          868
LLLOCEOS        866 IN PROCEDURE LLMAIN          868
LLADVANCE       866 IN PROCEDURE LLMAIN          868
```

Figure 15-4 continued: LDRA Testbed Data Flow Analysis of LL_COMPILE

```
1================================================
                TYPE DD ANOMALIES
 VARIABLE     DEFINE                          REDEFINE
 ================================================
 LLTOP          598                             594
 LLSTACK.LASTCHILD
                669                             671
 LLSTACK.TOP    669                             671
                                  ...
 LLSTACK.ATTRIBUTE
                784                             785
 LLSTACK.DATA   784                             785
 LLSTACK.LASTCHILD
                785                             786
 LLSTACK.TOP    785                             786
 LLSTACK.PARENT
                785                             786
 LLSTACK.ATTRIBUTE
                785                             786
 LLSTACK.DATA   785                             786
 LLTOP          827 IN PROCEDURE EXPAND         841 IN PROCEDURE ERASE
 LLADVANCE      797                             827 IN PROCEDURE EXPAND
 LLADVANCE      841 IN PROCEDURE ERASE          797
 LLADVANCE      797                             821 IN PROCEDURE TESTSYNCH


1==========================
 PROCEDURE PARAMETER ANALYSIS
 ==========================
 ----------------------------
 PROCEDURE LLFIND
      PARAMETER ITEM         IS SOMETIMES REFERENCED INSIDE THE PROCEDURE
      PARAMETER WHICH        IS SOMETIMES REFERENCED INSIDE THE PROCEDURE
 ----------------------------
 PROCEDURE LLPRTSTRING
      PARAMETER STR          IS ALWAYS REFERENCED INSIDE THE PROCEDURE
 ----------------------------
 PROCEDURE LLPRTTOKEN
      DOES NOT HAVE ANY PARAMETERS
 ----------------------------
                                  ...
 ----------------------------
 PROCEDURE GET_CHARACTER
      PARAMETER EOS          IS ALWAYS DEFINED INSIDE THE PROCEDURE
      PARAMETER NEXT         IS SOMETIMES REFERENCED
                                     AND DEFINED INSIDE THE PROCEDURE
      PARAMETER MORE         ***** IS NOT USED IN THE PROCEDURE *****
 ----------------------------
                                  ...
```

Figure 15-4 continued: LDRA Testbed Data Flow Analysis of LL_COMPILE

```
Path Analysis
-------------
                1   paths in procedure        LL_COMPILE
                9   paths in procedure        LLFIND
                4   paths in procedure        LLPRTSTRING
                2   paths in procedure        LLPRTTOKEN
                1   paths in procedure        LLSKIPTOKEN
                1   paths in procedure        LLSKIPNODE
                1   paths in procedure        LLSKIPBOTH
                1   paths in procedure        LLFATAL
                3   paths in procedure        GET_CHARACTER
               35   paths in procedure        MAKE_TOKEN
                2   paths in procedure        LLNEXTTOKEN
                1   paths in procedure        LLMAIN
                3   paths in procedure        CVT_STRING
               20   paths in procedure        READGRAM
               34   paths in procedure        PARSE
               63   paths in procedure        BUILDRIGHT
                2   paths in procedure        BUILDSELECT
                4   paths in procedure        ERASE
               14   paths in procedure        EXPAND
                3   paths in procedure        TESTSYNCH
                6   paths in procedure        MATCH
               51   paths in procedure        SYNCHRONIZE

Information Flow - Variable Dependency Results
----------------------------------------------

...

In Procedure LLFIND
-------------------

Strongly defined variables:
---------------------------

  Variable                  Strongly Dependent          Weakly Dependent
-------------------------------------------------------------------------
  HIGH                      'GLOBAL'LLTABLESIZE
-------------------------------------------------------------------------
  LOW                                                   'GLOBAL'LLTABLESIZE
-------------------------------------------------------------------------

Weakly defined variables:
-------------------------

  Variable                  Strongly Dependent          Weakly Dependent
-------------------------------------------------------------------------
  MIDPOINT                  'GLOBAL'LLTABLESIZE
-------------------------------------------------------------------------
```

Figure 15-5. LDRA Testbed Information Flow Analysis for LLFIND

```
STRUCTURED PROGRAMMING VERIFICATION WILL USE THE FOLLOWING  7 STRUCTURES
-----------------------------------------------------------------------

SIMPLE COLLAPSE
REPEAT LOOP
CASE
WHILE DO
IF THEN
IF THEN ELSE
FOR LOOP
1

        ********************************************************
        ********************************************************
        **                                                    **
        **                                                    **
        **             COMPLEXITY ANALYSIS FOR                **
        **                                                    **
        **                                                    **
        **                PROCEDURE LLFIND                    **
        **                                                    **
        **                                                    **
        ********************************************************
        ********************************************************

LIST OF KNOTS
-------------

=======================================================================
 FROM   TO     FROM   TO      DOWN-DOWN  UP-DOWN     UP-UP
=======================================================================
  155  172     163  869          T
  155  172     165  869          T
  155  172     171  155                     T
  157  160     159  170          T
  159  170     163  869          T
  159  170     165  869          T
  161  169     163  869          T
  161  169     165  869          T
  161  169     167  170          T
  162  165     163  869          T
  162  165     164  166          T
  163  869     171  155                     T
  164  166     165  869          T
  165  869     171  155                     T
-----------------------------------------------------------------------
TOTAL NUMBER OF KNOTS      =   14

NUMBER OF DOWN-DOWN KNOTS =   11
NUMBER OF UP-DOWN KNOTS   =    3
NUMBER OF UP-UP KNOTS     =    0
```

Figure 15-6. LDRA Testbed Complexity Analysis for LLFIND

```
BASIC BLOCK DISPLAY
-------------------
BRANCH FROM LINE  163 JUMPS OUT OF PROCEDURE
THIS ANALYSIS WILL TREAT IT AS IF IT GOES TO LINE  174,
IMMEDIATELY AFTER THE END OF THE PROCEDURE

BRANCH FROM LINE  165 JUMPS OUT OF PROCEDURE
THIS ANALYSIS WILL TREAT IT AS IF IT GOES TO LINE  174,
IMMEDIATELY AFTER THE END OF THE PROCEDURE
```

LINE NUMBER	STATEMENT	
146	function LLFIND(ITEM: LLSTRINGS; WHICH: LLSTYLE) return	1
147	INTEGER is	1
	...	
150	LOW, MIDPOINT, HIGH: INTEGER;	1
151		1
152	begin	1
153	LOW := 1;	1
154	HIGH := LLTABLESIZE + 1;	1
155	while LOW /= HIGH loop	2
156	MIDPOINT := (HIGH + LOW) / 2;	3
157	if ITEM < LLSYMBOLTABLE(MIDPOINT).KEY then	3
158	HIGH := MIDPOINT;	4
159	elsif	4
160	ITEM = LLSYMBOLTABLE(MIDPOINT).KEY	5
161	then	5
162	if LLSYMBOLTABLE(MIDPOINT).KIND = WHICH then	6
163	return(MIDPOINT);	7
164	else	8
165	return(0);	9
166	end if;	10
167	else	10
168	-- ITEM > LLSYMBOLTABLE(MIDPOINT).KEY	10
169	LOW := MIDPOINT + 1;	11
170	end if;	12
171	end loop;	12
172	return(0); -- item is not in table	13
173	end LLFIND;	13

Figure 15-6 continued: LDRA Testbed Complexity Analysis for LLFIND

```
AVERAGE LENGTH OF BASIC BLOCK =  1.62 LINES

BLOCK     8 IS UNREACHABLE - REMOVE FROM FURTHER CONSIDERATION
BLOCK    10 IS UNREACHABLE - REMOVE FROM FURTHER CONSIDERATION

PROCEDURE ENTRY AT BASIC BLOCK     1
PROCEDURE EXIT  AT BASIC BLOCK    14


KNOTS
-----
```

FROM BLOCK	TO BLOCK	FROM BLOCK	TO BLOCK	FROM LINE	TO LINE	FROM LINE	TO LINE
2	13	7	14	155	172	163	174
2	13	9	14	155	172	165	174
2	13	12	2	155	172	171	155
3	5	4	12	157	160	159	170
4	12	7	14	159	170	163	174
4	12	9	14	159	170	165	174
5	11	7	14	161	169	163	174
5	11	9	14	161	169	165	174
6	9	7	14	162	165	163	174
7	14	12	2	163	174	171	155
9	14	12	2	165	174	171	155

```
NUMBER OF BLOCK CONNECTIONS  =   15
NUMBER OF BLOCKS             =   12

COMPLEXITY MEASURE OF MCCABE =    5
************************************

NUMBER OF KNOTS              =   11
************************************


1INTERVAL ANALYSIS
------------------

INTERVALS OF ORDER    1
-------------------------
HEADER BLOCK       1
INTERVAL BLOCKS    1

HEADER BLOCK       2
INTERVAL BLOCKS    2    3   13    4    5    6   11    7    9   12   14

NUMBER OF INTERVALS =    2
AVERAGE LENGTH OF INTERVAL =  6.00 BLOCKS
```

Figure 15-6 continued: LDRA Testbed Complexity Analysis for LLFIND

```
INTERVALS OF ORDER     2
-----------------------
HEADER BLOCK        1
INTERVAL BLOCKS     1     2

NUMBER OF INTERVALS =     1
AVERAGE LENGTH OF INTERVAL =  2.00 BLOCKS

PROCEDURE REDUCIBLE IN THE INTERVAL SENSE
*****************************************


1STRUCTURED PROGRAMMING VERIFICATION
------------------------------------

----- CONNECTION DISPLAY -----
BLOCK     1 CONNECTS TO BLOCKS     2
BLOCK     2 CONNECTS TO BLOCKS     3    13
BLOCK     3 CONNECTS TO BLOCKS     4     5
BLOCK     4 CONNECTS TO BLOCKS    12
BLOCK     5 CONNECTS TO BLOCKS     6    11
BLOCK     6 CONNECTS TO BLOCKS     7     9
BLOCK     7 CONNECTS TO BLOCKS    14
BLOCK     9 CONNECTS TO BLOCKS    14
BLOCK    11 CONNECTS TO BLOCKS    12
BLOCK    12 CONNECTS TO BLOCKS     2
BLOCK    13 CONNECTS TO BLOCKS    14


THE LINES OF CODE CONTAINED IN EACH BLOCK ARE
BLOCK     1 CONTAINS LINES  146 TO  154    -   BLOCK     1
BLOCK     2 CONTAINS LINES  155 TO  155    -   BLOCK     2
BLOCK     3 CONTAINS LINES  156 TO  157    -   BLOCK     3
BLOCK     4 CONTAINS LINES  158 TO  159    -   BLOCK     4
BLOCK     5 CONTAINS LINES  160 TO  161    -   BLOCK     5
BLOCK     6 CONTAINS LINES  162 TO  162    -   BLOCK     6
BLOCK     7 CONTAINS LINES  163 TO  163    -   BLOCK     7
BLOCK     9 CONTAINS LINES  165 TO  165    -   BLOCK     9
BLOCK    11 CONTAINS LINES  169 TO  169    -   BLOCK    11
BLOCK    12 CONTAINS LINES  170 TO  171    -   BLOCK    12
BLOCK    13 CONTAINS LINES  172 TO  173    -   BLOCK    13
BLOCK    14 CONTAINS LINES  174 TO  174    -   BLOCK    14

----- STRUCTURED CODE FOUND -----
IF THEN ELSE          FORMED OF BLOCKS     6     7     9    14
      THE FOLLOWING BLOCKS ARE MERGED      6     7     9
```

Figure 15-6 continued: LDRA Testbed Complexity Analysis for LLFIND

```
----- CONNECTION DISPLAY -----
BLOCK     1 CONNECTS TO BLOCKS     2
BLOCK     2 CONNECTS TO BLOCKS     3    13
BLOCK     3 CONNECTS TO BLOCKS     4     5
BLOCK     4 CONNECTS TO BLOCKS    12
BLOCK     5 CONNECTS TO BLOCKS     6    11
BLOCK     6 CONNECTS TO BLOCKS    14
BLOCK    11 CONNECTS TO BLOCKS    12
BLOCK    12 CONNECTS TO BLOCKS     2
BLOCK    13 CONNECTS TO BLOCKS    14

THE LINES OF CODE CONTAINED IN EACH BLOCK ARE
BLOCK     1 CONTAINS LINES  146 TO  154    -   BLOCK     1
BLOCK     2 CONTAINS LINES  155 TO  155    -   BLOCK     2
BLOCK     3 CONTAINS LINES  156 TO  157    -   BLOCK     3
BLOCK     4 CONTAINS LINES  158 TO  159    -   BLOCK     4
BLOCK     5 CONTAINS LINES  160 TO  161    -   BLOCK     5
BLOCK     6 CONTAINS LINES  162 TO  163    -  BLOCKS     6 TO     7
                      AND   165 TO  165    -   BLOCK     9
BLOCK    11 CONTAINS LINES  169 TO  169    -   BLOCK    11
BLOCK    12 CONTAINS LINES  170 TO  171    -   BLOCK    12
BLOCK    13 CONTAINS LINES  172 TO  173    -   BLOCK    13
BLOCK    14 CONTAINS LINES  174 TO  174    -   BLOCK    14

NO FURTHER STRUCTURE FOUND


KNOTS
-----
```

FROM BLOCK	TO BLOCK	FROM BLOCK	TO BLOCK	FROM LINE	TO LINE	FROM LINE	TO LINE
2	13	6	14	155	172	162	174
2	13	12	2	155	172	171	155
3	5	4	12	157	160	159	170
4	12	6	14	159	170	162	174
5	11	6	14	161	169	162	174
6	14	12	2	162	174	171	155

```
ESSENTIAL KNOTS =     6
************************

ESSENTIAL COMPLEXITY OF MCCABE =     4
****************************************

PROCEDURE NOT STRUCTURED
**************************
```

Figure15-6 continued: LDRA Testbed Complexity Analysis for LLFIND

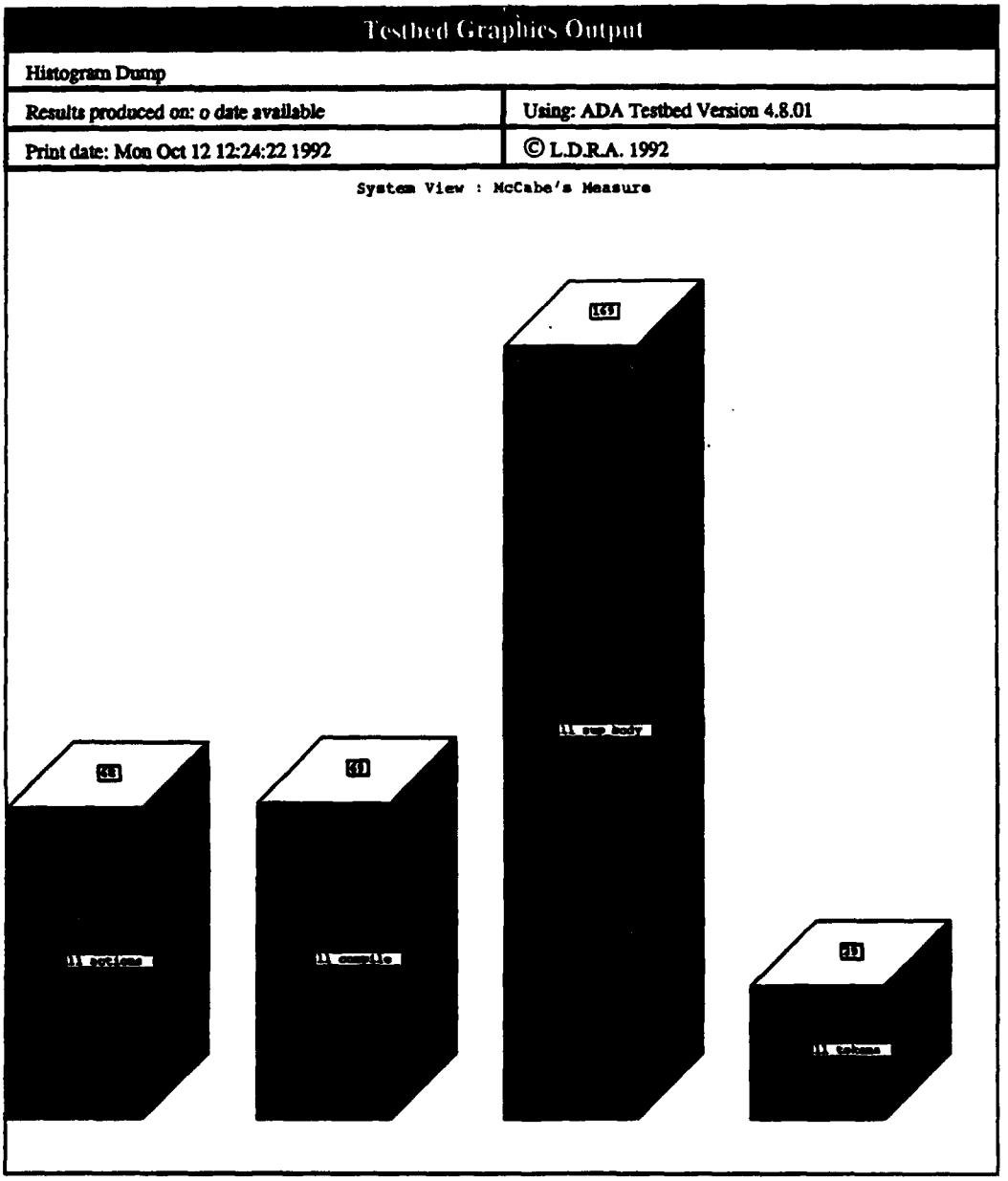


Figure 15-7. LDRA Testbed System View McCabe's Complexity Measure

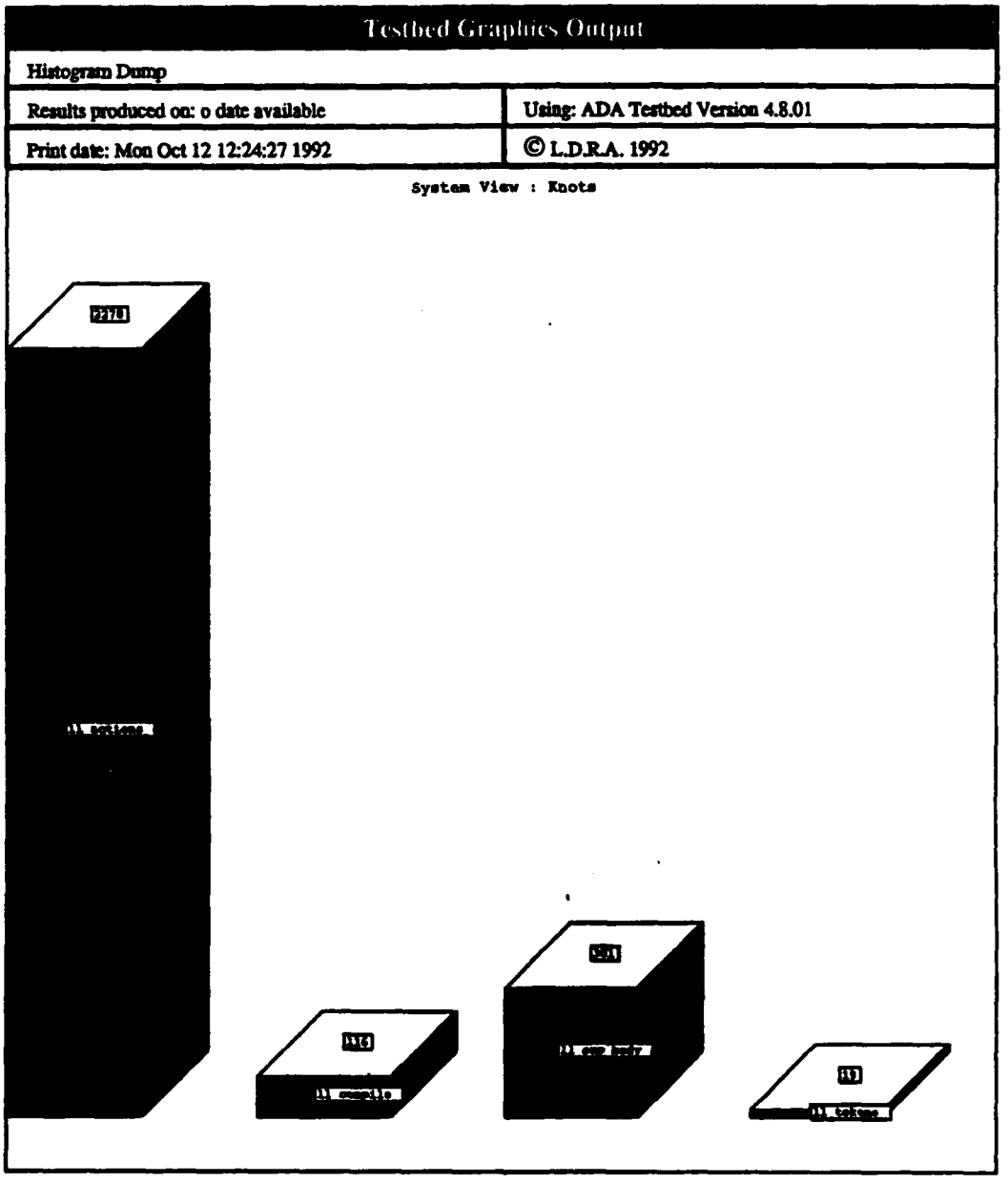


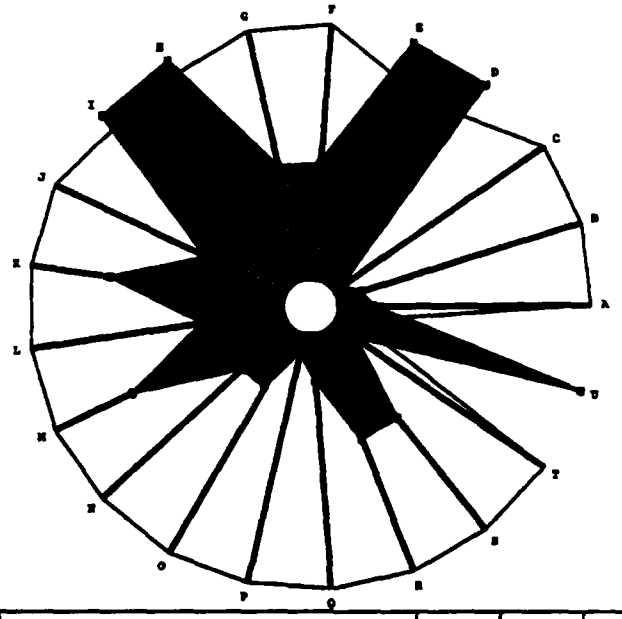
Figure 15-8. LDRA Testbed System View Knots Complexity Measure

Testbed Graphics Output

Kiviat Diagram

Results produced on: Mon Oct 12 11:53:31 1992	Using: ADA Testbed Version 4.8.01
Print date: Mon Oct 12 12:30:35 1992	© L.D.R.A. 1992

Kiviat Diagram



Label	Bar	Lmb	Upb	Val
A	Path Fragment (LCSAJ) Coverage	0.30	1.00	0.68
B	Branch Coverage	0.40	1.00	0.66
C	Statement Coverage	0.50	1.00	0.52
D	Average Number of Essential Knots in a procedure	0.00	2.00	2.36
E	Total Number of Essential Knots in Program (c)'	0	44	52
F	Average Essential McCabes Measure in a procedure	1.00	3.00	1.91
G	Total Essential McCabes Measure in Program (c)	1	45	21
H	Average Number of Knots in a procedure	1.00	5.00	5.27
I	Total Number of Knots in Program (c)	22	110	116
J	Average McCabes Measure in a procedure	1.00	10.00	4.09
K	Total McCabes Measure in Program	1	100	69
L	Average Maximum Interval Nesting in a procedure	1.00	3.00	1.64
M	Total Maximum Interval Nesting in Program	1	4	3
N	Average number of Order 1 Intervals in a procedure	1.00	5.00	2.00
O	Number of Order 1 Intervals in program (c)	22	110	44
P	Average number of Basic Blocks in a procedure	1.00	100.00	8.32
Q	Number of Basic Blocks in program	1	1000	183
R	Average Number of Lines of Code in a procedure	5.00	50.00	25.00
S	Number of Lines of Code in program (c)	110	1100	550
T	Number of Procedures in Program	1	200	22
U	Number of Static Penalties in Program	0	10	126

Figure 15-9. LDRA Testbed Kiviat Graph for LLFIND

```
DETERMINATION OF LINEAR CODE SEQUENCE AND JUMP TRIPLES
------------------------------------------------------

=============================================================================
START  FINISH   LINE
LABEL   LABEL  NUMBER          STATEMENT
=============================================================================
-----------------------------------------------------------------------------
                                ...
       FINISH    145                                                      1
-----------------------------------------------------------------------------
START            146  function  LLFIND(ITEM: LLSTRINGS; WHICH: LLSTYLE) return 3
                 147           INTEGER  is                                3
                 148     --  Find item in symbol table -- return index or 0 if not found
                 149     --  Assumes symbol table is sorted in ascending order.3
                 150     LOW, MIDPOINT, HIGH: INTEGER;                    3
                 151                                                      3
                 152  begin                                               3
                 153     LOW := 1;                                        3
                 154     HIGH := LLTABLESIZE + 1;                         3
START  FINISH    155     while  LOW /= HIGH  loop                         6
                 156        MIDPOINT := (HIGH + LOW) / 2;                 4
       FINISH    157        if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then  4
                 158           HIGH := MIDPOINT;                          2
       FINISH    159        elsif                                         2
-----------------------------------------------------------------------------
START            160              ITEM = LLSYMBOLTABLE(MIDPOINT).KEY        10
       FINISH    161                                                then   10
       FINISH    162           if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then  9
       FINISH    163              return( MIDPOINT );                      8
-----------------------------------------------------------------------------
                 164           else                                        0 UNREACE
-----------------------------------------------------------------------------
START  FINISH    165              return( 0 );                             8
-----------------------------------------------------------------------------
                 166           end if;                                     0 UNREACE
                 167        else                                           0 UNREACE
-----------------------------------------------------------------------------
START            168              --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY    1
                 169           LOW := MIDPOINT + 1;                       1
START            170        end if;                                       2
       FINISH    171     end loop;                                        2
-----------------------------------------------------------------------------
START  FINISH    172     return( 0 );  -- item is not in table            8
-----------------------------------------------------------------------------
                 173  end  LLFIND;                                        0 UNREACE
-----------------------------------------------------------------------------
START            174                                                      1
       FINISH    175                                                      1
-----------------------------------------------------------------------------
START            176  procedure  LLPRTSTRING( STR: LLSTRINGS )  is        3
                                ...
```

Figure 15-10. LDRA Testbed LCSAJ Analysis for LL_COMPILE

```
1LINEAR CODE SEQUENCE AND JUMP TRIPLES
--------------------------------------

--------------------
START   FINISH   JUMP TO
 LINE    LINE     LINE
--------------------
   31      145      174
  146      155      172
  146      157      160
  146      159      170
                              ...
  166      167      170     UNREACHABLE *****
  168      171      155
  170      171      155
  172      172      327
  172      172      332
  172      172      335
  172      172      341
  172      172      344
  172      172      350
  172      172      790
  172      172      793
  174      175      189
  176      181      187
  176      184      187
  176      186      181
                           ...
                           ...
  861      861      868
  862      867      413
  868      868       -1

NUMBER OF LCSAJS IN PROGRAM =  327      (     1 UNREACHABLE)

BRANCH FROM  164 TO  166 IS UNREACHABLE
BRANCH FROM  167 TO  170 IS UNREACHABLE
BRANCH FROM  270 TO  666 IS UNREACHABLE
BRANCH FROM  270 TO  771 IS UNREACHABLE
BRANCH FROM  270 TO  851 IS UNREACHABLE
BRANCH FROM  753 TO  756 IS UNREACHABLE
BRANCH FROM  758 TO  775 IS UNREACHABLE
```

Figure 15-10 continued: LDRA Testbed LCSAJ Analysis for LL_COMPILE

```
ATTRIBUTE CODES
---------------
  L     LOCAL VARIABLE
  G     GLOBAL VARIABLE
  P     PARAMETER
  LG    LOCAL VARIABLE USED AS GLOBAL IN OTHER PROCEDURE
================================================================================

                                    ...

PROCEDURE LLFIND
START LINE   146 END LINE   174

CALLS NO PROCEDURES

IS CALLED BY THE FOLLOWING PROCEDURES
NAME          CALLED ON LINE
----          --------------
MAKE_TOKEN      325   330   333   339   342   348
PARSE           788   791

VARIABLE USAGE INFORMATION
NAME       ATTRIB  OCCURS ON LINE
----       ------  --------------
'GLOBAL'LLTABLESIZE
               G    154
HIGH           L    150   154   155   156   158
ITEM           P    146   157   160
LLSYMBOLTABLE
               G    157   160   162
LOW            L    150   153   155   156   169
MIDPOINT       L    150   156   157   158   160   162   163   169
WHICH          P    146   162
================================================================================

                                    ...

====================
MANAGEMENT SUMMARY
====================

THE FOLLOWING VARIABLES HAVE ONLY ONE OCCURRENCE
NAME          OCCURS ON LINE
----          --------------

                                    ...

LLCURTOK.TABLEINDEX  405
'GLOBAL'LLSTRINGS    308
'GLOBAL'IN_FILE      539
'GLOBAL'LLSTRINGLENGTH 542
LLCURTOK.ATTRIBUTE   804
TABLEINDEX           453
I                    529
'GLOBAL'STANDARD_ERROR 661
LLCURTOK.TABLEINDEX  625
LLCURTOK.PRINTVALUE  724
```

Figure 15-11. LDRA Testbed Cross Reference Analysis for LLFIND

```
                    DYNAMIC COVERAGE ANALYSIS REPORT
                    --------------------------------

 PRODUCED BY LDRA SOFTWARE TESTBED: DYNAMIC COVERAGE ANALYSER

 DYNAMIC COVERAGE ANALYSIS REPORT OPTIONS SELECTED
 -------------------------------------------------
 PROCEDURE BY PROCEDURE PRINTOUT FOR ALL PROCEDURES

 TRACING OPTIONS SELECTED
 ------------------------
 NO TRACE REQUESTED

 PROFILES INCLUDED FOR THE FOLLOWING TEST DATA SETS
 --------------------------------------------------
   1) test1.lex
   2) sample.lex

1**************************************************
 ***** THE FOLLOWING PROCEDURE(S) WERE ENTERED *****
 ***** UNEXPECTEDLY. USUAL CAUSE IS-           *****
 *****    A) MISSING LEVEL                     *****
 *****    B) ANALYSIS OF MODULE WITH NO MASTER *****
 ***** THIS MAY CAUSE LOCAL TRACE AND STATEMENT*****
 ***** EXECUTION PROFILE TO BE INCORRECT       *****
 **************************************************
 CALL PROCEDURE (MISSING LEVEL) LLFIND
 RETURNING FROM LLFIND
                                    ...
 CALL PROCEDURE (MISSING LEVEL) LLFIND
 RETURNING FROM LLFIND

1MISSING LINEAR CODE SEQUENCE AND JUMP TRIPLES
 ---------------------------------------------
 ========================================
 START   FINISH  JUMPTO
  LINE    LINE    LINE             COUNT
 ========================================
   160     163     398               130
   160     163     399                90
   165     165     398                 2
                                    ...

1MISSING BRANCHES
 ----------------
 ============================
 FROM     TO
  LINE    LINE          COUNT
 ============================
   163     398            130
   163     399             90
   165     398              2
                           ...
```

Figure 15-12. LDRA Testbed Dynamic Analysis for LL_COMPILE

```
1*****************************************************
*****************************************************
**                                                 **
**              DYNAMIC ANALYSIS FOR               **
**               PROCEDURE  LLFIND                 **
**                                                 **
*****************************************************
*****************************************************

1STATEMENT EXECUTION PROFILE
----------------------------
```

LINE NUMBER	STATEMENT
146	function LLFIND(ITEM: LLSTRINGS; WHICH: LLSTYLE) return
147	INTEGER is
148	-- Find item in symbol table -- return index or 0 if not found.
149	-- Assumes symbol table is sorted in ascending order.
150	LOW, MIDPOINT, HIGH: INTEGER;
151	
152	begin
153	LOW := 1;
154	HIGH := LLTABLESIZE + 1;
155	while LOW /= HIGH loop
156	MIDPOINT := (HIGH + LOW) / 2;
157	if ITEM < LLSYMBOLTABLE(MIDPOINT).KEY then
158	HIGH := MIDPOINT;
159	elsif
160	ITEM = LLSYMBOLTABLE(MIDPOINT).KEY
161	then
162	if LLSYMBOLTABLE(MIDPOINT).KIND = WHICH then
163	return(MIDPOINT);
164	else
165	return(0);
166	end if;
167	else
168	-- ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
169	LOW := MIDPOINT + 1;
170	end if;
171	end loop;
172	return(0); -- item is not in table
173	end LLFIND;

SUMMARY	OLD COUNT	NEW COUNT	TOTAL
NUMBER OF EXECUTABLE LINES	17	17	17
NUMBER EXECUTED	16	17	17
NUMBER NOT EXECUTED	1	0	0
TEST EFFECTIVENESS RATIO 1	0.94	1.00	1.00

Figure 15-12 continued: LDRA Testbed Dynamic Analysis for LL_COMPILE

1BRANCH EXECUTION PROFILE

FROM LINE	TO LINE	OLD COUNT	NEW COUNT	TOTAL
155	156	898	1422	2320
155	172	63	88	151
157	158	429	646	1075
157	160	469	776	1245
159	170	429	646	1075
161	162	135	224	359
161	168	334	552	886
162	163	135	222	357
162	165	0 ****	2	2
163	327	0 ****	0 ****	0 ****
163	332	0 ****	0 ****	0 ****
163	335	0 ****	0 ****	0 ****
163	341	0 ****	0 ****	0 ****
163	344	0 ****	0 ****	0 ****
163	350	0 ****	0 ****	0 ****
163	790	1	1	2
163	793	1	1	2
165	327	0 ****	0 ****	0 ****
165	332	0 ****	0 ****	0 ****
165	335	0 ****	0 ****	0 ****
165	341	0 ****	0 ****	0 ****
165	344	0 ****	0 ****	0 ****
165	350	0 ****	0 ****	0 ****
165	790	0 ****	0 ****	0 ****
165	793	0 ****	0 ****	0 ****
171	155	763	1198	1961
172	327	0 ****	0 ****	0 ****
172	332	0 ****	0 ****	0 ****
172	335	0 ****	0 ****	0 ****
172	341	0 ****	0 ****	0 ****
172	344	0 ****	0 ****	0 ****
172	350	0 ****	0 ****	0 ****
172	790	0 ****	0 ****	0 ****
172	793	0 ****	0 ****	0 ****

SUMMARY	OLD COUNT	NEW COUNT	TOTAL
NUMBER OF BRANCHES	34	34	34
NUMBER EXECUTED	11	12	12
NUMBER NOT EXECUTED	23	22	22
TEST EFFECTIVENESS RATIO 2	0.32	0.35	0.35

Figure 15-12 continued: LDRA Testbed Dynamic Analysis for LL_COMPILE

```
THE FOLLOWING BRANCHES HAVE NOT BEEN EXECUTED BY ANY TEST DATA SET
------------------------------------------------------------------

( 163, 327)  ( 163, 332)  ( 163, 335)  ( 163, 341)  ( 163, 344)  ( 163, 350)  ( 165, 3
( 165, 332)  ( 165, 335)  ( 165, 341)  ( 165, 344)  ( 165, 350)  ( 165, 790)  ( 165, 7
( 172, 327)  ( 172, 332)  ( 172, 335)  ( 172, 341)  ( 172, 344)  ( 172, 350)  ( 172, 7
( 172, 793)

MISSING BRANCHES
----------------

==============================
FROM      TO
 LINE     LINE              COUNT
==============================
  163      398                130
  163      399                 90
  165      398                  2
  172      398                 88

1LINEAR CODE SEQUENCE AND JUMP EXECUTION PROFILE
------------------------------------------------

=============================================================================
START    FINISH   JUMPTO              OLD              NEW
 LINE     LINE     LINE              COUNT            COUNT            TOTAL
=============================================================================
  146      155      172                0 ****           0 ****           0 ****
  155      155      172               63               88              151
  146      157      160               22               38               60
  155      157      160              447              738             1185
  146      159      170              176              274              450
  155      159      170              253              372              625
  160      161      168              334              552              886
  160      162      165                0 ****           2                2
  160      163      327                0 ****           0 ****           0 ****
  160      163      332                0 ****           0 ****           0 ****
  160      163      335                0 ****           0 ****           0 ****
  160      163      341                0 ****           0 ****           0 ****
  160      163      344                0 ****           0 ****           0 ****
  160      163      350                0 ****           0 ****           0 ****
  160      163      790                1                1                2
  160      163      793                1                1                2
  165      165      327                0 ****           0 ****           0 ****
  165      165      332                0 ****           0 ****           0 ****
  165      165      335                0 ****           0 ****           0 ****
  165      165      341                0 ****           0 ****           0 ****
  165      165      344                0 ****           0 ****           0 ****
  165      165      350                0 ****           0 ****           0 ****
  165      165      790                0 ****           0 ****           0 ****
  165      165      793                0 ****           0 ****           0 ****
```

Figure 15-12 continued: LDRA Testbed Dynamic Analysis for LL_COMPILE

168	171	155	334	552	886
170	171	155	429	646	1075
172	172	327	0 ****	0 ****	0 ****
172	172	332	0 ****	0 ****	0 ****
172	172	335	0 ****	0 ****	0 ****
172	172	341	0 ****	0 ****	0 ****
172	172	344	0 ****	0 ****	0 ****
172	172	350	0 ****	0 ****	0 ****
172	172	790	0 ****	0 ****	0 ****
172	172	793	0 ****	0 ****	0 ****

SUMMARY	OLD COUNT	NEW COUNT	TOTAL
NUMBER OF LCSAJS	34	34	34
NUMBER EXECUTED	10	11	11
NUMBER NOT EXECUTED	24	23	23
TEST EFFECTIVENESS RATIO 3	0.29	0.32	0.32

THE FOLLOWING LCSAJS HAVE NOT BEEN EXECUTED BY ANY TEST DATA SET

(146, 155, 172)	(160, 163, 327)	(160, 163, 332)	(160, 163, 335)	(160, 163,
(160, 163, 344)	(160, 163, 350)	(165, 165, 327)	(165, 165, 332)	(165, 165,
(165, 165, 341)	(165, 165, 344)	(165, 165, 350)	(165, 165, 790)	(165, 165,
(172, 172, 327)	(172, 172, 332)	(172, 172, 335)	(172, 172, 341)	(172, 172,
(172, 172, 350)	(172, 172, 790)	(172, 172, 793)		

MISSING LINEAR CODE SEQUENCE AND JUMP TRIPLES

START LINE	FINISH LINE	JUMPTO LINE	COUNT
160	163	398	130
160	163	399	90
165	165	398	2
172	172	398	88

1STATEMENT EXECUTION HISTORY SUMMARY

	EXECUTABLE STATEMENTS	NUMBER EXECUTED OLD	NUMBER EXECUTED NEW	NUMBER EXECUTED TOTAL	TER 1 OLD	TER 1 NEW	TER 1 TOTAL
LL_COMPILE	27	27	27	27	1.00	1.00	1.00
LLFIND	17	16	17	17	0.94	1.00	1.00
LLPRTSTRING	10	0	0	0	0.00	0.00	0.00
LLPRTTOKEN	10	0	0	0	0.00	0.00	0.00
LLSKIPTOKEN	10	0	0	0	0.00	0.00	0.00

Figure 15-12 continued: LDRA Testbed Dynamic Analysis for LL_COMPILE

LLSKIPNODE	12	0	0	0	0.00	0.00	0.00
LLSKIPBOTH	13	0	0	0	0.00	0.00	0.00
LLFATAL	8	0	0	0	0.00	0.00	0.00
GET_CHARACTER	14	0	0	0	0.00	0.00	0.00
MAKE_TOKEN	67	0	0	0	0.00	0.00	0.00
LLNEXTTOKEN	8	8	8	8	1.00	1.00	1.00
LLMAIN	14	14	14	14	1.00	1.00	1.00
CVT_STRING	9	0	0	0	0.00	0.00	0.00
READGRAM	38	38	38	38	1.00	1.00	1.00
PARSE	73	56	56	56	0.77	0.77	0.77
BUILDRIGHT	62	54	54	54	0.87	0.87	0.87
BUILDSELECT	8	8	8	8	1.00	1.00	1.00
ERASE	11	11	11	11	1.00	1.00	1.00
EXPAND	43	38	38	38	0.88	0.88	0.88
TESTSYNCH	14	0	0	0	0.00	0.00	0.00
MATCH	15	13	13	13	0.87	0.87	0.87
SYNCHRONIZE	57	0	0	0	0.00	0.00	0.00
TOTAL	540	283	284	284	0.52	0.53	0.53

1SUB-CONDITIONS SUMMARY

	NUMBER EXECUTED			TER CON		
SUB-CONDITIONS	OLD	NEW	TOTAL	OLD	NEW	TOTAL
LL_COMPILE PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLFIND PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLPRTSTRING PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLPRTTOKEN PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLSKIPTOKEN PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLSKIPNODE PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLSKIPBOTH PROCEDURE CONTAINS NO SUB-CONDITIONS						
LLFATAL PROCEDURE CONTAINS NO SUB-CONDITIONS						
GET_CHARACTER PROCEDURE CONTAINS NO SUB-CONDITIONS						
MAKE_TOKEN PROCEDURE CONTAINS NO SUB-CONDITIONS						
...						
EXPAND	8	8	8	1.00	1.00	1.00
TESTSYNCH PROCEDURE CONTAINS NO SUB-CONDITIONS						
MATCH	4	4	4	1.00	1.00	1.00
SYNCHRONIZE	12	0	0	0.00	0.00	0.00
TOTAL	24	12	12	0.50	0.50	0.50

Figure 15-12 continued: LDRA Testbed Dynamic Analysis for LL_COMPILE

1BRANCH EXECUTION HISTORY SUMMARY

		NUMBER EXECUTED			TER 2		
	BRANCHES	OLD	NEW	TOTAL	OLD	NEW	TOTAL
LL_COMPILE	13	13	13	13	1.00	1.00	1.00
LLFIND	34	11	12	12	0.32	0.35	0.35
LLPRTSTRING	9	0	0	0	0.00	0.00	0.00
LLPRTTOKEN	9	0	0	0	0.00	0.00	0.00
LLSKIPTOKEN	2	0	0	0	0.00	0.00	0.00
LLSKIPNODE	3	0	0	0	0.00	0.00	0.00
LLSKIPBOTH	3	0	0	0	0.00	0.00	0.00
LLFATAL	2	0	0	0	0.00	0.00	0.00
GET_CHARACTER	7	0	0	0	0.00	0.00	0.00
MAKE_TOKEN	31	0	0	0	0.00	0.00	0.00
LLNEXTTOKEN	5	4	4	4	0.80	0.80	0.80
LLMAIN	5	5	5	5	1.00	1.00	1.00
CVT_STRING	7	0	0	0	0.00	0.00	0.00
READGRAM	20	20	20	20	1.00	1.00	1.00
PARSE	39	25	25	25	0.64	0.64	0.64
BUILDRIGHT	26	22	22	22	0.85	0.85	0.85
BUILDSELECT	4	4	4	4	1.00	1.00	1.00
ERASE	7	7	7	7	1.00	1.00	1.00
EXPAND	18	15	15	15	0.83	0.83	0.83
TESTSYNCH	11	0	0	0	0.00	0.00	0.00
MATCH	11	7	7	7	0.64	0.64	0.64
SYNCHRONIZE	26	0	0	0	0.00	0.00	0.00
TOTAL	292	133	134	134	0.46	0.46	0.46

1LCSAJ EXECUTION HISTORY SUMMARY

		NUMBER EXECUTED			TER 3		
	LCSAJS	OLD	NEW	TOTAL	OLD	NEW	TOTAL
LL_COMPILE	13	13	13	13	1.00	1.00	1.00
LLFIND	34	10	11	11	0.29	0.32	0.32
LLPRTSTRING	10	0	0	0	0.00	0.00	0.00
LLPRTTOKEN	13	0	0	0	0.00	0.00	0.00
LLSKIPTOKEN	2	0	0	0	0.00	0.00	0.00

Figure 15-12 continued: LDRA Testbed Dynamic Analysis for LL_COMPILE

LLSKIPNODE	3	0	0	0	0.00	0.00	0.00
LLSKIPBOTH	3	0	0	0	0.00	0.00	0.00
LLFATAL	2	0	0	0	0.00	0.00	0.00
GET_CHARACTER	6	0	0	0	0.00	0.00	0.00
MAKE_TOKEN	33	0	0	0	0.00	0.00	0.00
LLNEXTTOKEN	7	3	3	3	0.43	0.43	0.43
LLMAIN	5	5	5	5	1.00	1.00	1.00
CVT_STRING	9	0	0	0	0.00	0.00	0.00
READGRAM	25	20	20	20	0.80	0.80	0.80
PARSE	34	23	23	23	0.68	0.68	0.68
BUILDRIGHT	32	24	24	24	0.75	0.75	0.75
BUILDSELECT	5	4	4	4	0.80	0.80	0.80
ERASE	8	7	7	7	0.88	0.88	0.88
EXPAND	20	15	15	15	0.75	0.75	0.75
TESTSYNCH	12	0	0	0	0.00	0.00	0.00
MATCH	12	6	6	6	0.50	0.50	0.50
SYNCHRONIZE	38	0	0	0	0.00	0.00	0.00
TOTAL	326	130	131	131	0.40	0.40	0.40

1SUMMARY OF EFFECT OF CURRENT TEST DATA SET ON THE COVERAGE METRICS

PROCEDURE NAME	TER 1	TER 2	TER 3
LL_COMPILE	1.00	1.00	1.00
LLFIND	Increased	Increased	Increased
LLPRTSTRING	No Change	No Change	No Change
LLPRTTOKEN	No Change	No Change	No Change
LLSKIPTOKEN	No Change	No Change	No Change
LLSKIPNODE	No Change	No Change	No Change
LLSKIPBOTH	No Change	No Change	No Change
LLFATAL	No Change	No Change	No Change
GET_CHARACTER	No Change	No Change	No Change
MAKE_TOKEN	No Change	No Change	No Change
LLNEXTTOKEN	1.00	No Change	No Change
LLMAIN	1.00	1.00	1.00
CVT_STRING	No Change	No Change	No Change
READGRAM	1.00	1.00	No Change
PARSE	No Change	No Change	No Change
BUILDRIGHT	No Change	No Change	No Change
BUILDSELECT	1.00	1.00	No Change
ERASE	1.00	1.00	No Change
EXPAND	No Change	No Change	No Change
TESTSYNCH	No Change	No Change	No Change
MATCH	No Change	No Change	No Change
SYNCHRONIZE	No Change	No Change	No Change

Figure 15-12 continued: LDRA Testbed Dynamic Analysis for LL_COMPILE

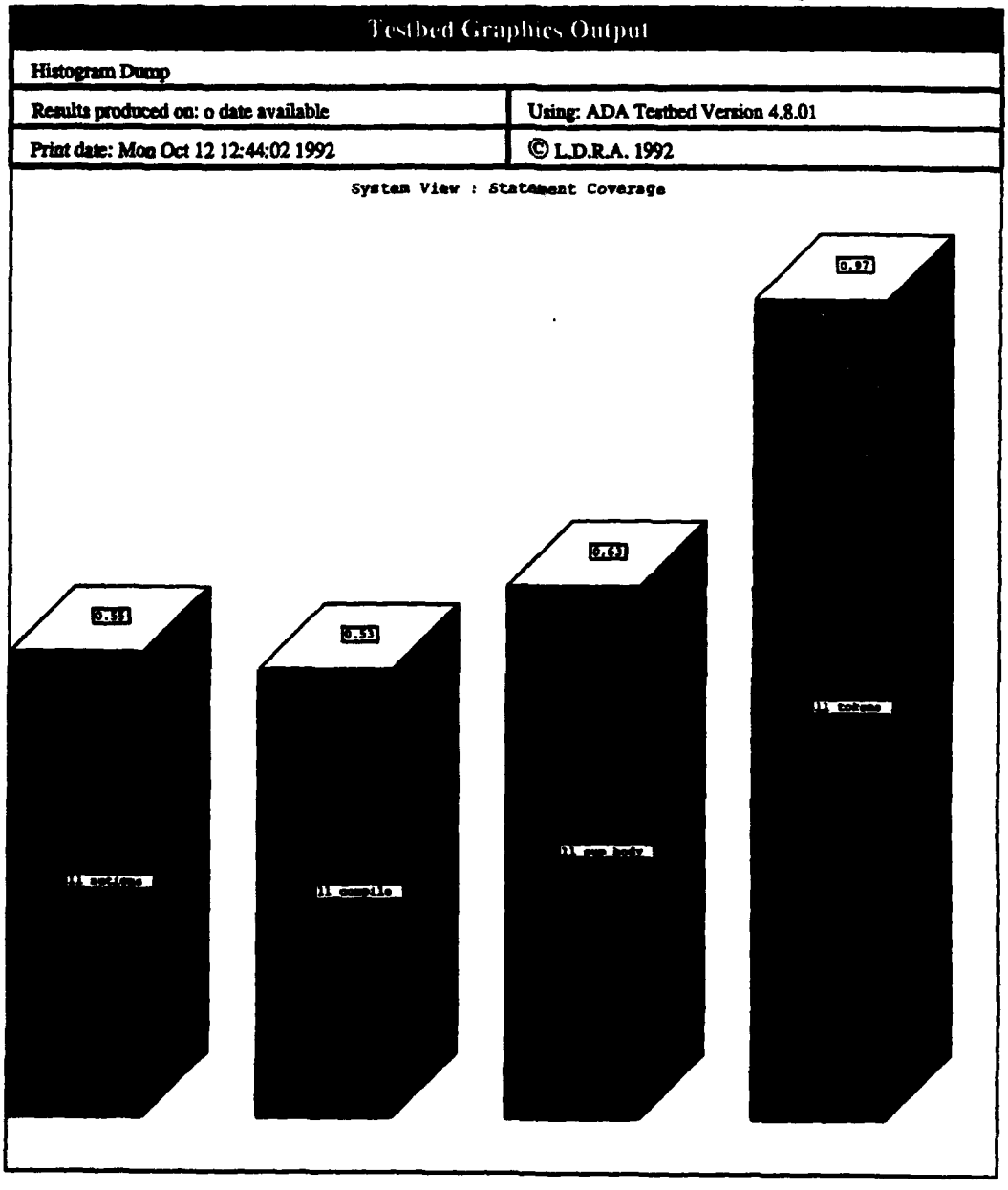


Figure 15-13. LDRA Testbed System View Statement Coverage

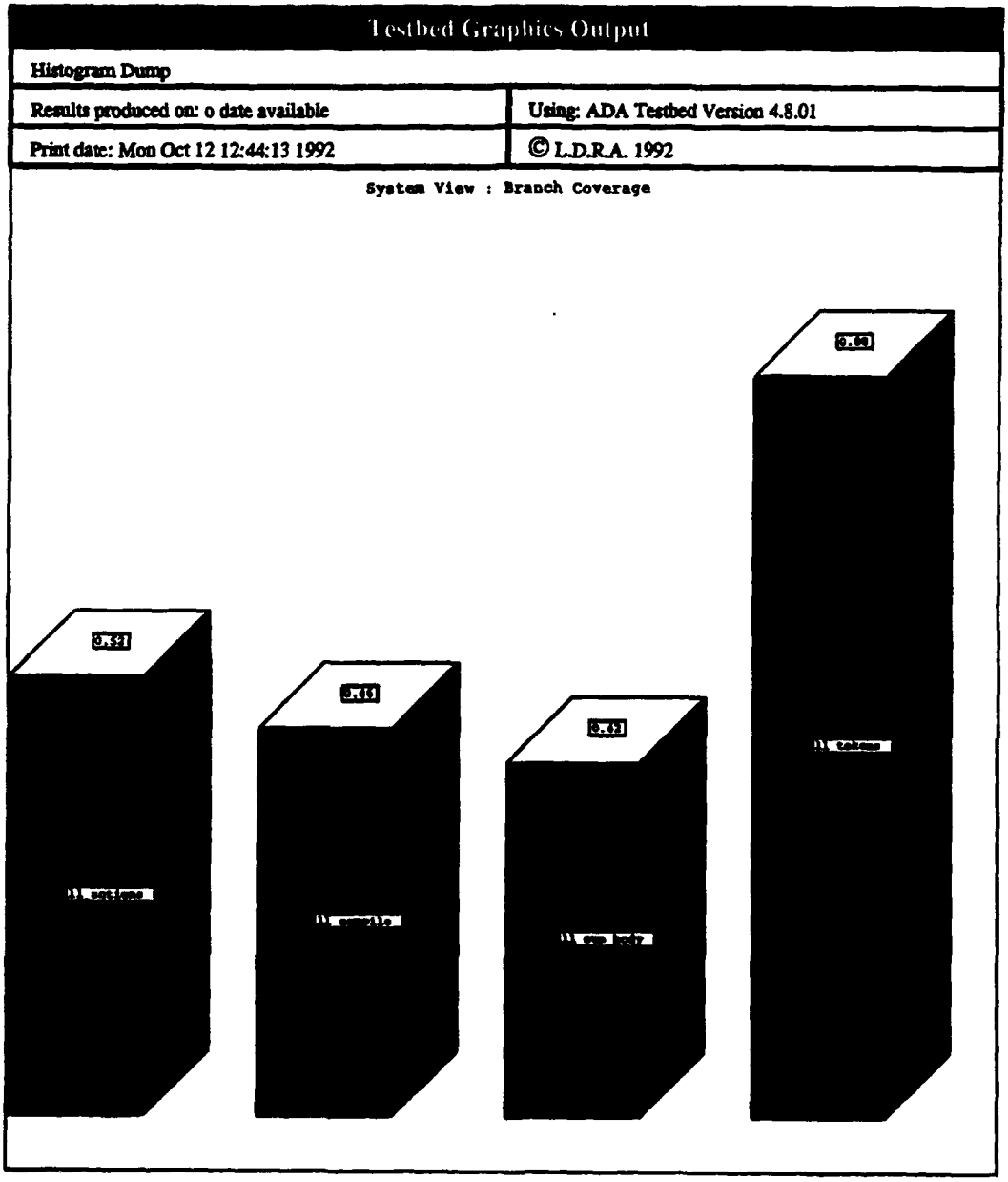


Figure 15-14. LDRA Testbed System View Branch Coverage

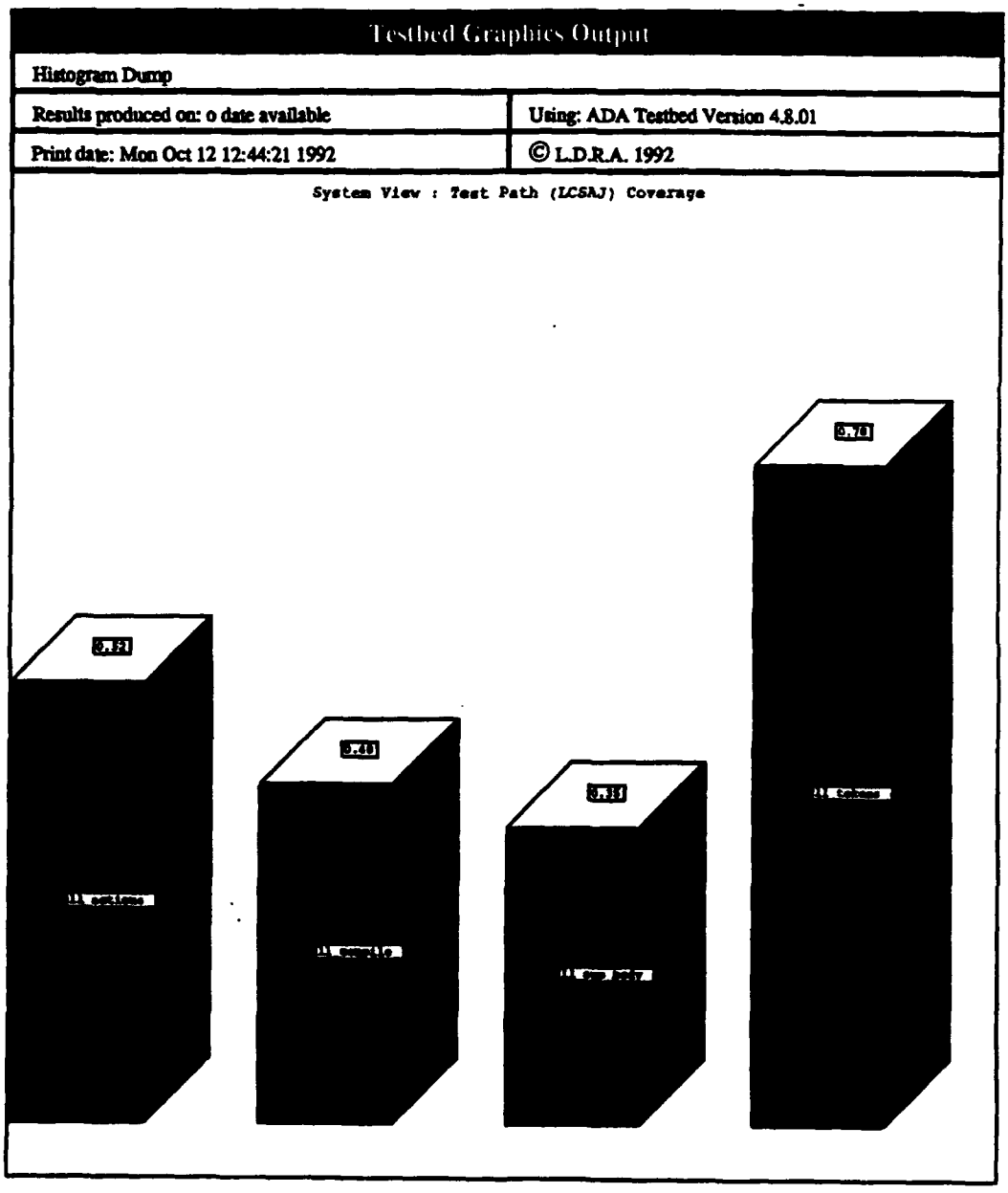


Figure 15-15. LDRA Testbed System View Test Path (LCSAJ) Coverage

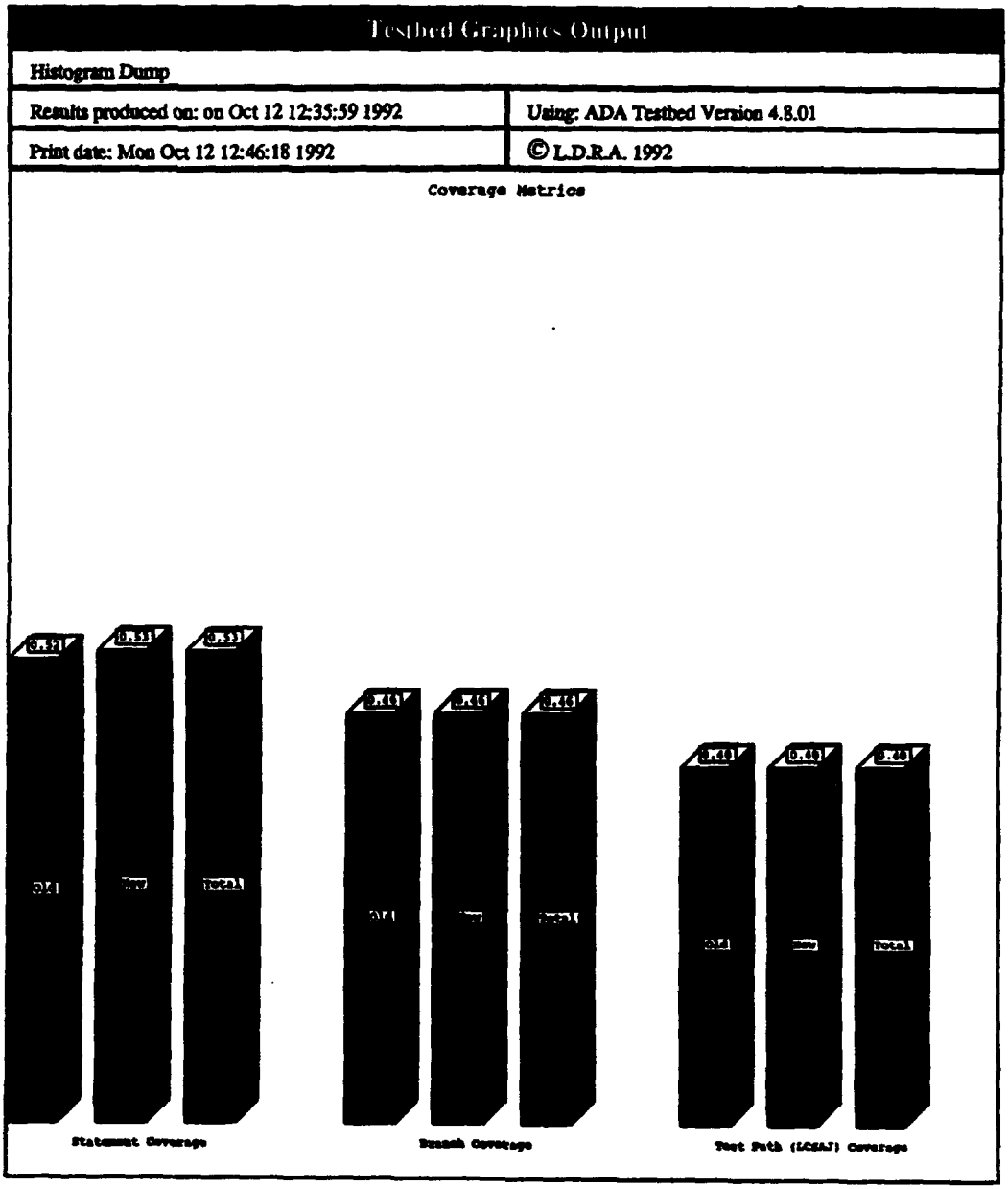


Figure 15-16. LDRA Testbed Coverage Achieved Comparison

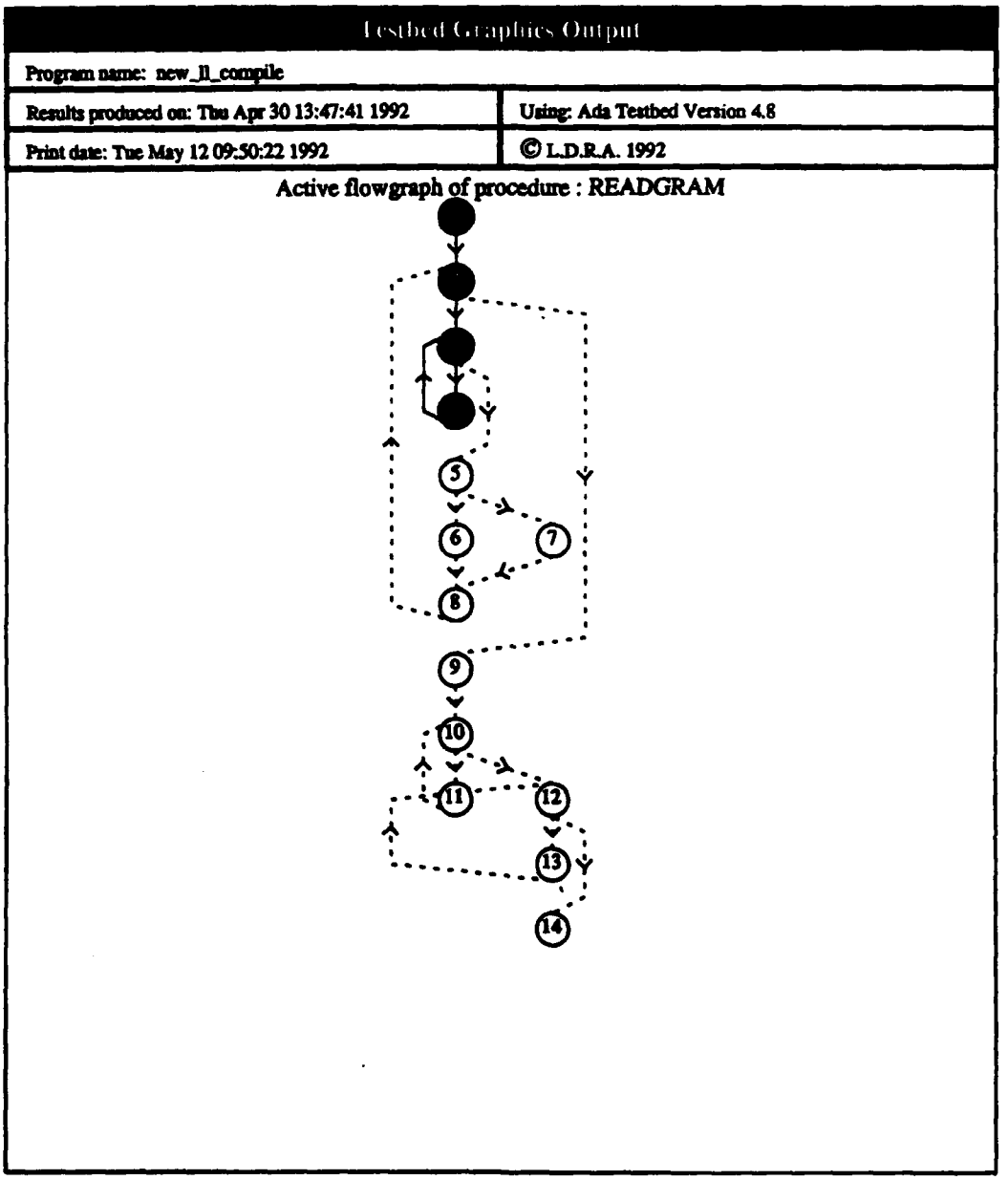


Figure 15-17. LDRA Testbed Active Flowgraph of READGRAM

```
1
          ******************************************************
          ******************************************************
          **                                                  **
          **              DATA SET ANALYSIS FOR               **
          **                                                  **
          **                PROCEDURE LLFIND                  **
          **                                                  **
          ******************************************************
          ******************************************************

1TEST DATA SET ANALYSIS
 ---------------------

 ================================================================================
   LINE           DATA SETS USED
 ================================================================================
    146  test1.lex
         sample.lex

                                        ...

    162  test1.lex
         sample.lex

    163  test1.lex
         sample.lex

    164   ****

    165  sample.lex

    166   ****

    167   ****

    168  test1.lex
         sample.lex

    169  test1.lex
         sample.lex

    170  test1.lex
         sample.lex

    171  test1.lex
         sample.lex

    172  test1.lex
         sample.lex

    173   ****
```

Figure 15-18. LDRA Testbed Data Set Analysis for LLFIND

```
1PROFILE ANALYSIS
 ----------------


 ----------------
 LIST OF DATA SETS
 ----------------

    1)  test1.lex
    2)  sample.lex

 ---------------------------------
 ANALYSIS OF EACH DATA SET IN TURN
 ---------------------------------
 DATA SET      1 CONTRIBUTES NOTHING. SIZE =   44900
 DATA SET      2 IS NECESSARY         SIZE =   72092


 DATASET RECOMMENDED FOR REMOVAL IS     1
```

Figure 15-19. LDRA Testbed Profile Analysis

16. Logiscope

Logiscope employs the RADC quality metrics model to provide analysis of a set of user-tailorable quality metrics at both the unit and integration levels. It provides coverage analysis of statement blocks, branches, and LCSAJs at the unit level, and procedure-to-procedure path coverage analysis at the integration level. Additional capabilities include the generation of control and call graphs, structure analysis, and pseudo-code generation to support re-engineering.

Logiscope is one element of a comprehensive suite of CASE tools. AGE/ASA is a CASE tool supporting functional specification activities. Based on IDEF0 and finite state machine specification methods, it supports simulation and various static analyses including complexity analysis. It also provides test scenario generation for automatic production of functional test suites which can be fed into the simulator or used during code acceptance testing to ensure compliance with requirements. Scenario coverage can be measured during simulation. Support for design is available through AGE/GEODE. This tool is based on the Consulting Committee on International Telegraph and Telephone (CCITT) standardized language Specification and Description Language (SDL) and provides for design and simulation of real-time software with automatic code and application generation. AGE/GEODE also provides test process generation to allow independently testing the coherence of a process with respect to the rest of the design prior to system integration. A new tool, VEDA, supports simulation and validation of protocols specified in the International Organization for Standardization (ISO) standard language Estelle. Finally, DocBuilder is used to produce software documentation that can be configured to such standards as DoD-STD-2167A. It is based on the Standard Generalized Markup Language (SGML) ISO Standard 8879.

16.1 Tool Overview

Logiscope was developed by Verilog, a European company formed in 1984. It has been available since 1985 and there are over 5,000 users worldwide. Logiscope is marketed in the U.S. by Verilog, Inc., the U.S. subsidiary. This company also provides consulting and training services, and hot-line support for tool users.

Logiscope is available for over eighty languages and dialects, including Ada, C, C++, and Fortran. It is supported on a variety of workstations and mainframes under both Unix

and VMS, with graphic capabilities available through a number of windowing systems. As with all its tools, Verilog has focused on compatibility of Logiscope with international standards such as the Portable Common Tool Environment (PCTE), SDL, etc. Logiscope can be integrated with DecFuse, HP's SoftBench, and Software Back Plane. It supports host/target testing via use of a serial port between the host and target machines.

The evaluation was performed on Logiscope/Ada version 1.6.3. running on a Sun 4 workstation under UNIX and OpenWindows. At the time of evaluation, prices for Logiscope started at $14,000.

Logiscope consists of several parts:

- Analyzer. Processes source code to provide the data needed for the Results Editor.
- Results Editor. Takes the results file produced by the Analyzer and, potentially, the trace file produced by an instrumented program to generate various reports.
- Formatter. Compacts the execution trace produced by an instrumented program.
- Static Archiver. Gathers various analysis results and manages results obtained for different versions of the software.
- Dynamic Archiver. Accumulates results for a set of test runs and enables multiple test suite management.

While the Analyzer is unique to a particular programming language, the remaining tools are language independent. All tools operate in both interactive and batch mode.

The Analyzer operates in either static or dynamic mode, although application of Logiscope begins with static analysis of the software under test. The software should have previously been compiled and, where several compilation units are employed, these must be submitted in the compilation order (this restriction applies to Ada code only). In static mode, the Analyzer calculates the appropriate set of basic counts that will be used to assesses the quality of the software under examination. In dynamic mode, it instruments source code for instruction block, decision-to-decision path, LCSAJ coverage, or procedure-to-procedure coverage. Files are instrumented individually and, potentially, for different types of coverage measurement. Dynamic analysis also provides path and condition identification to aid test data generation. After instrumentation, the user compiles, links, and then executes instrumented source code as usual.

In general, the Analyzer can analyze files singly or as a group. It generates a Results File that the Results Editor uses to generate a variety of reports. There is a facility for combining separate Results Files together to form a single file for a subsystem, or system. It can search for such items as code based on keywords, or code that falls within certain values for a given metric or criteria.

The Results Editor also operates in static and dynamic modes, presenting results at different levels: details for each application component, a synthesis of component results for the entire application, and global application architecture information.

Quality analysis is the primary static analysis function and Logiscope employs the RADC quality metrics model to define quality measurement at three levels of abstraction. At the lowest level of the model there are thirty five predefined *primitive metrics*. The user can define upper and lower bounds for these metrics to allow Logiscope to flag out-of-bounds values. (Verilog provides default values for these bounds that are based on their experience over time.) The user can specify algorithms to weight and combine the primitive metrics into up to fifteen *composite metrics*. Then higher-level *quality criteria* allow classifying components based on their computed quality values. These criteria can also be used to get an overall quality value for a module and report on final acceptance or rejection based on this value.

Logiscope distinguishes between unit-level metrics and architectural metrics. In the first case, McCabe's control-oriented measures are calculated, as well as Halstead's textually-oriented Software Science measures. At the architectural level, Logiscope uses Mohanty's metrics to calculate accessibility, testability, hierarchy complexity, structural complexity, system testability, call graph entropy, and the number of direct calls.

Quality results are displayed using the Results Editor in static mode. In addition to the Results File produced by the Analyzer, the editor requires a Reference File that contains the definitions of the metrics being used. (A different Reference File can be maintained for each project, allowing customization across development efforts.) For quality reporting at the component level, the user can request Kiviat diagrams to show achieved metrics values with respect to the defined limit values. These diagrams are used to display up to 30 user-selected metrics, graphically showing those metrics that fall out-of-bounds. Metrics can be displayed by component, or as a statistical average over a group of components. Kiviat diagrams can also be segmented into quadrants to provide an additional layer of abstraction. Criteria graphs are available to display information relative to all associations between metrics and criteria, while showing the situation of metrics with respect to limit values. These graphs also specify the category to which the component belongs.

At the global level, histograms of metrics distributions and criteria distributions are available. Additionally, when there is a large number of components, the user can request a graphical distribution for a particular interval or a distribution of components as a function of the limit values defined in the quality model.

Finally, a Quality Report uses the components' classification based on the quality criteria to present a summary in the form of the percentage of components within the set of limit values. This report assesses whether quality recommendations for a given criteria have been met, or computes a statistical average over a group of components.

Also in static mode, the Results Editor generates control graphs to provide insight into component structure and behavior, and call graphs to describe the calling relationships of analyzed components. Control graphs can be annotated with either source or pseudocode line numbers. Logiscope supports control graph exploration with a zoom capability and the display of a reduced or structured form of a control graph. The reduced form can be used to verify that a program meets the requirements of structured programming and identify elements that do not conform. The principle of control graph reduction consists of representing as a single node the control structures that have only one input and one output. The most deeply nested structures are reduced at each successive reduction stage, and the user can terminate this process when desired. Alternatively, the structured view displays the underlying structures expressed in combinations of if-then-else statements and branch statements to reveal the hidden structuring of the processing. Measurements of a set of intrinsic characteristics are available for initial, reduced, and structured control graphs. This allows comparing the set of alternative, equivalent views of a complex control graph and can help a user to determine how to improve the program structure.

Exploration of call graphs is also provided to support the identification of critical components at the architectural level, and of design rules that have been violated. This is achieved by display of partial views and manipulation of call graphs, and quality evaluation. A call graph can be displayed from any root, and the display limited to a view of the root's descendants, ascendants, or both. Components can be grouped to clarify, for example, which components can call that set. A call graph can be limited to display of the Logiscope analyzed components alone.

Before the Results Editor can be used in dynamic mode for coverage reporting, the trace file produced by the instrumented program must be formatted. Subsequently, the editor can report on the achieved coverage at both component and global levels. At unit level, the user can request detailed reports on instruction block, decision-to-decision path, and LCSAJ coverage. For each type of coverage this includes a listing identifying each instance of the instruction block, decision-to-decision path, or LCSAJ unit, supported by the conditions required to execute that instance as appropriate, and whether or not it was executed. A path list also indicates program paths that have not been executed. Unit coverage results can be

annotated on dynamic control graphs to provide easy assessment of the completeness of unit testing. Histograms of the distribution of components as a function of coverage rate are available for rapid assessment of coverage progression throughout testing. These histograms are accompanied by a distribution list that shows the coverage achieved for each component. This distribution list shows the number of times each test case exercised each coverage instance and can be used to determine how well particular test cases support or duplicate each other.

The Dynamic Archiver is used to group the results obtained for a series of tests to allow reporting on cumulative test coverage. Here formatted trace files are grouped into named test suites that are stored in archive files. The Results Editor can then be run on an archive file to generate, for example, distribution histograms for the accumulated instruction block, decision-to-decision path, and LCSAJ coverage for all components.

At the global level, the editor reports on procedure-to-procedure path coverage. Here coverage results can be annotated on call graphs to provide quick insight into the completeness of integration testing. An accompanying textual report details the calling and called relationships for each procedure-to-procedure path and whether that path was executed. An additional report, the coverage table, identifies the particular paths invoked by each test case.

16.2 Observations

Ease of use. The Results Editor provides on-line help with a list of available commands and command descriptions. Components can be grouped into a workspace to facilitate operating on a set of components as a whole. A broad selection of graphical output formats is available, including histograms, tables, and pie charts.

Logiscope provides the user with considerable flexibility in defining the quality characteristics that should be assessed and reported. It comes with a series of default quality models, one for each of five different programming language. These can be used as is, the user can tailor them to his needs, or develop his own quality model from scratch.

Documentation and user support. The documentation is extensive and easy to follow. Verilog provided excellent user support.

Instrumentation overhead. Full instrumentation of the Ada Lexical Analyzer Generator (all components except ll_support) gave a size increase of just over 50%.

Ada restrictions. Pragmas are not processed by the Analyzer. It is not possible to measure the coverage of a *terminate* alternative in a selective wait.

Problems encountered. The Analyzer reported an error when analyzing one component (ll_support) of the Ada Lexical Analyzer Generator and this prevented instrumenting this component. Initially, some problems were encountered with reporting on LCSAJ coverage.

16.3 Planned Additions

Version 3.2 of Logiscope with a Common OSF/MOTIF graphical user interface was released in fall 1992. This new version is menu driven and supports navigation between source code (or pseudocode) and graphs. Multiple, integrated windows are simultaneously available to provide a user with multiple perspectives of a single software component. In this new version, Logiscope is integrated with DocBuilder to provide automatic documentation of new or existing code. Meanwhile, Verilog is working to integrate Logiscope with various configuration management tools.

A companion tool which focuses on data flow analysis rather than control flow analysis is under development.

16.4 Sample Outputs

Figures 16-1 through 16-28 provide sample outputs from Logiscope.

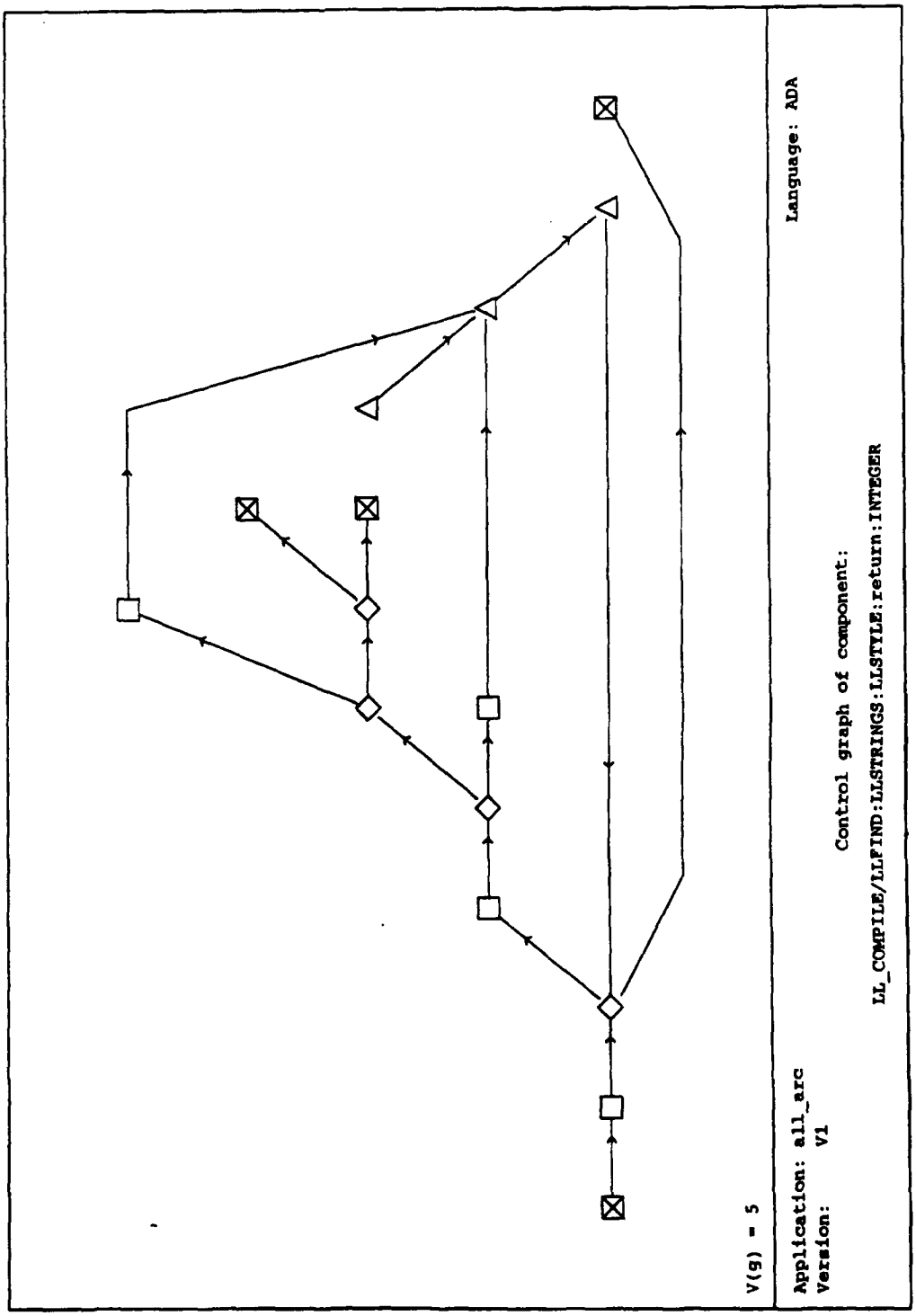


Figure 16-1. Logiscope Control Graph of Function LLFIND

```
Begin
 2 ADA_statement(s);
 While LOW /= HIGH Do
   1 ADA_statement(s);
   If ITEM < LLSYMBOLTABLE(MIDPOINT).KEY Then
     1 ADA_statement(s);
   Elsif not (ITEM < LLSYMBOLTABLE(MIDPOINT).KEY) and (ITEM =
LLSYMBOLTABLE(MIDPOINT).KEY) Then
     If LLSYMBOLTABLE(MIDPOINT).KIND = WHICH Then
       Exit of Subprogram;
     Else
       Exit of Subprogram;.
     End If;
   Else
     1 ADA_statement(s);
   End If;
 End of While;
End;
```

```
Text of component:

LL_COMPILE/LLFIND:LLSTRINGS:LLSTYLE:return:INTEGER

Application: all_arc
Version:     V1
Language:    ADA
File     : ll_compile.a
```

Figure 16-2. Logiscope Textual Representation of Control Graph of Function LLFIND

Basic counts of component :
LL_COMPILE/LLFIND:LLSTRINGS:LLSTYLE:return:INTEGER

Number of statements	11	Number of comments	2
Number of labels	0	Number of jump statements	0
Total number of operators	29	Number of different operators	13
Total number of operands	28	Number of different operands	11
Total number of calls	0	Number of different calls	0

Operators	Nbr	Operators	Nbr
( ) exp	4	=	2
( ) tab	3	ELSE	2
+	3	ELSIF THEN	1
/	1	IF THEN .. END IF	2
/=	1	RETURN	3
:=	5	WHILE LOOP .. END LOOP	1
<	1		

Operands	Nbr	Operands	Nbr
0	2	LLSYMBOLTABLE(.).KIND	1
1	3	LLTABLESIZE	1
2	1	LOW	4
HIGH	4	MIDPOINT	7
ITEM	2	WHICH	1
LLSYMBOLTABLE(.).KEY	2		

Figure 16-3. Logiscope Basic Counts for Function LLFIND

```
                                    ...
 21 with LL_DECLARATIONS, INTEGER_TEXT_IO, TEXT_IO;
 22
 23 procedure LL_COMPILE is
                                    ...
161   function LLFIND (ITEM : LLSTRINGS; WHICH : LLSTYLE) return INTEGER is
162   --  Find item in symbol table -- return index or 0 if not found.
163   --  Assumes symbol table is sorted in ascending order.
164
165   LOW, MIDPOINT, HIGH : INTEGER;
166
167   begin                                                  (* DDP  1 Begin  *)
168
169     LOW := 1;
170     HIGH := LLTABLESIZE + 1;
171     while LOW /= HIGH loop                               (* DDP  2 While  *)
172        MIDPOINT := (HIGH + LOW) / 2;
173        if ITEM < LLSYMBOLTABLE (MIDPOINT).KEY then       (* DDP  3 If     *)
174            HIGH := MIDPOINT;
175        elsif ITEM = LLSYMBOLTABLE (MIDPOINT).KEY then    (* DDP  4 Else   *
176                                                          (* DDP  5 Else-If*)
177            if LLSYMBOLTABLE (MIDPOINT).KIND = WHICH then (* DDP  6 If     *
178                  return (MIDPOINT);
179            else                                          (* DDP  7 Else   *)
180                  return (0);
181            end if;
182        else                                              (* DDP  8 Else   *)
183        --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
184            LOW := MIDPOINT + 1;
185        end if;
186     end loop;                                            (* DDP  9 End-While *)
187     return (0);
188   -- item is not in table
189
190   end LLFIND;
191
192   procedure LLPRTSTRING (STR : LLSTRINGS) is
193     --  print non-blank prefix of str in quotes
194
195   begin                                                  (* DDP  1 Begin  *)
196
197     PUT (STANDARD_ERROR, '"');
198     for I in STR'range loop                              (* DDP  2 For_Loop*)
199        exit when STR(I) = ' ';                           (* DDP  3 If     *)
200                                                          (* DDP  4 Else   *)
201        PUT (STANDARD_ERROR, STR (I));
202     end loop;                                            (* DDP  5 End-For_Loop*
203     PUT (STANDARD_ERROR, '"');
204
205   end LLPRTSTRING;
                                    ...
852 end LL_COMPILE;
853
```

Figure 16-4. Logiscope Commented Listing for Function LLFIND

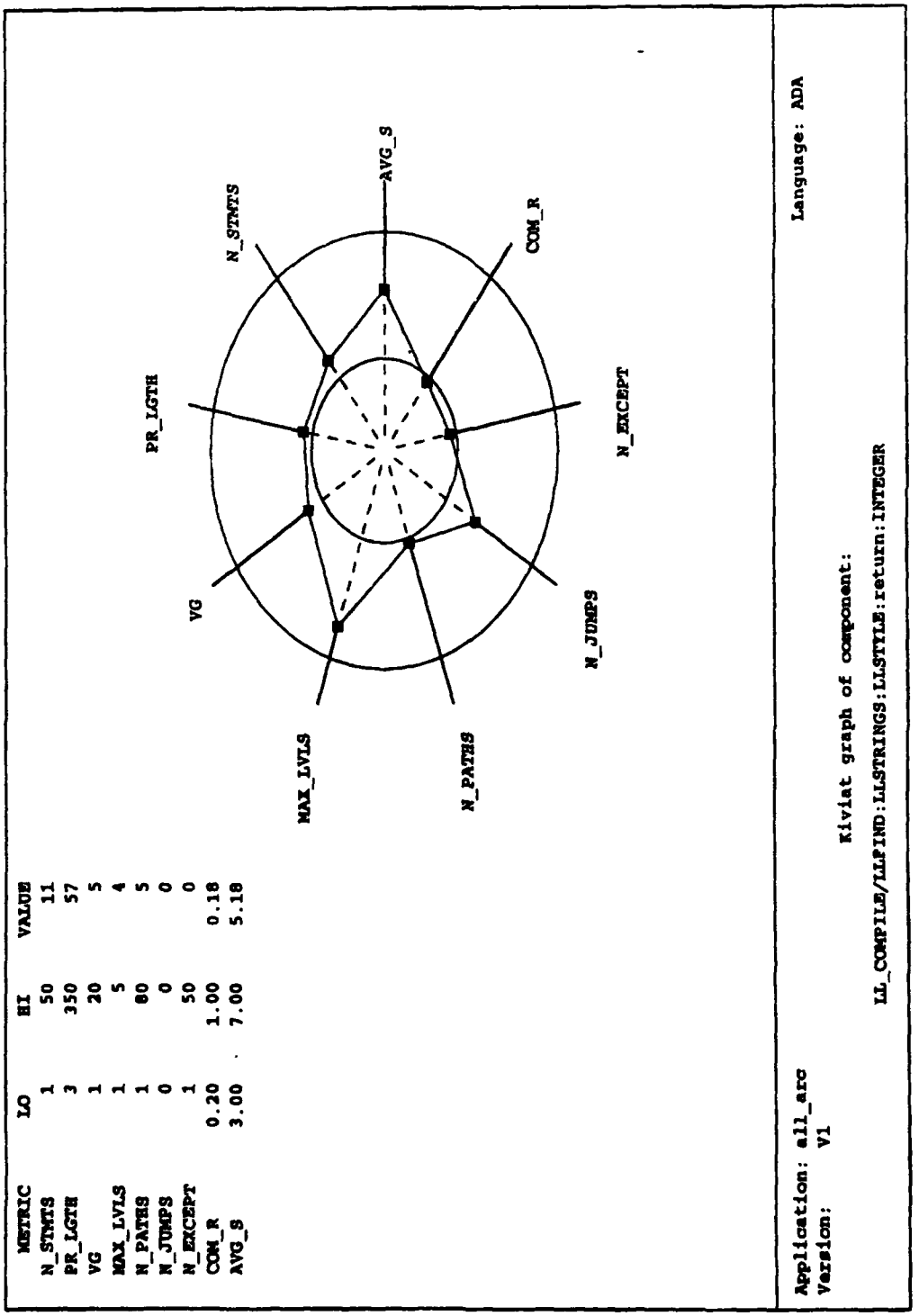


Figure 16-5. Logiscope Kiviat Graph of Function LLFIND

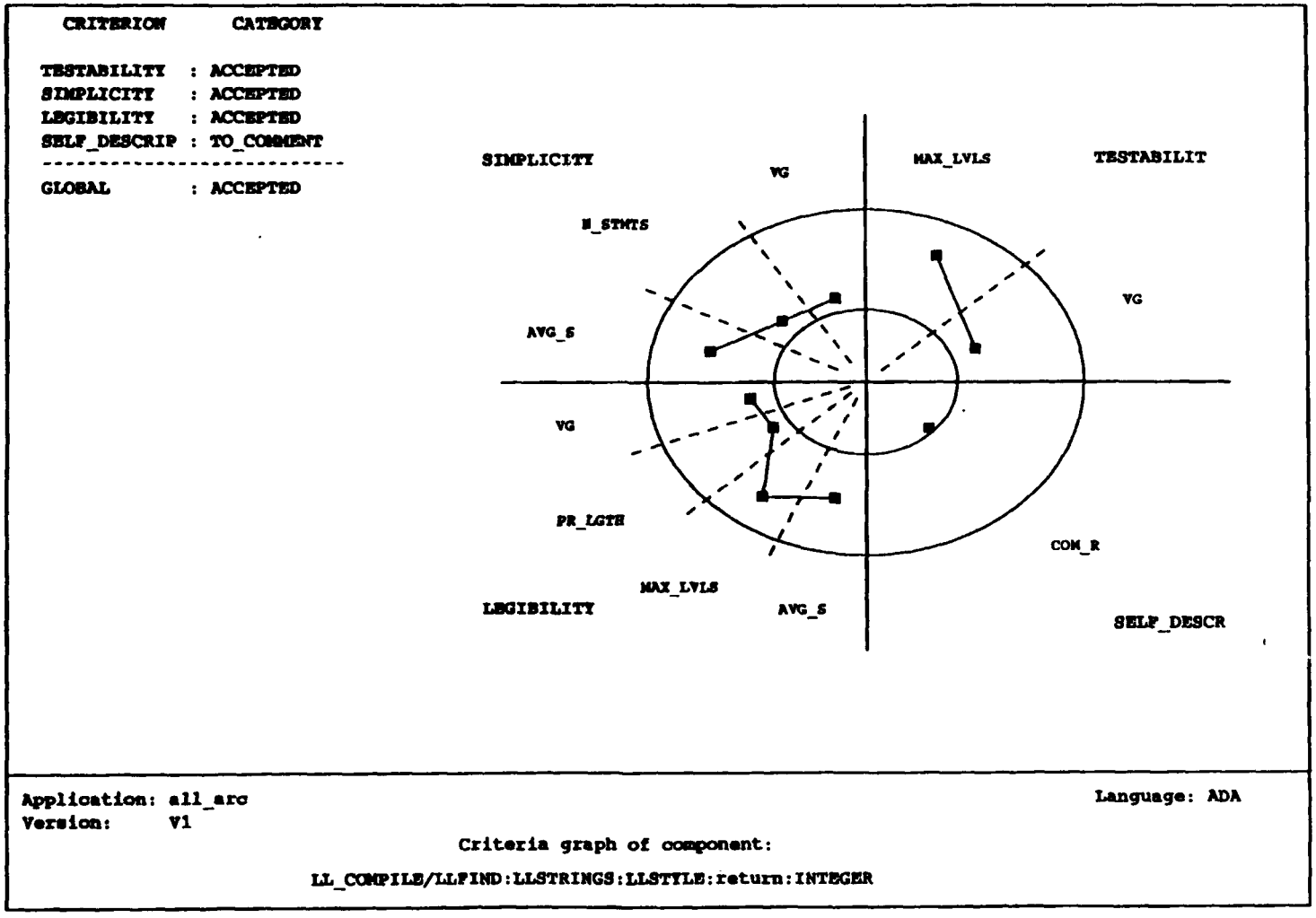
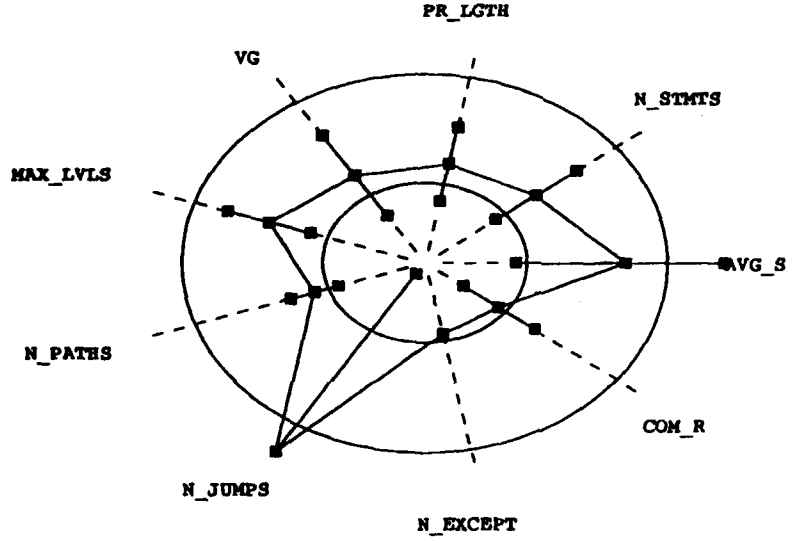


Figure 16-6. Logiscope Criteria Graph of Function LLFIND

METRIC	LO	HI	AVERAGE
N_STMTS	1	50	16.00
PR_LGTH	3	350	92.36
VG	1	20	6.01
MAX_LVLS	1	5	2.78
N_PATHS	1	80	9.66
N_JUMPS	0	0	0.12
N_EXCEPT	1	50	0.00
COM_R	0.20	1.00	0.16
AVG_S	3.00	7.00	5.80



Application: all_arc
Version: V1

Language: ADA

Kiviat graph of all components

66 Components

Figure 16-7. Logiscope Kiviat Graph of All Components

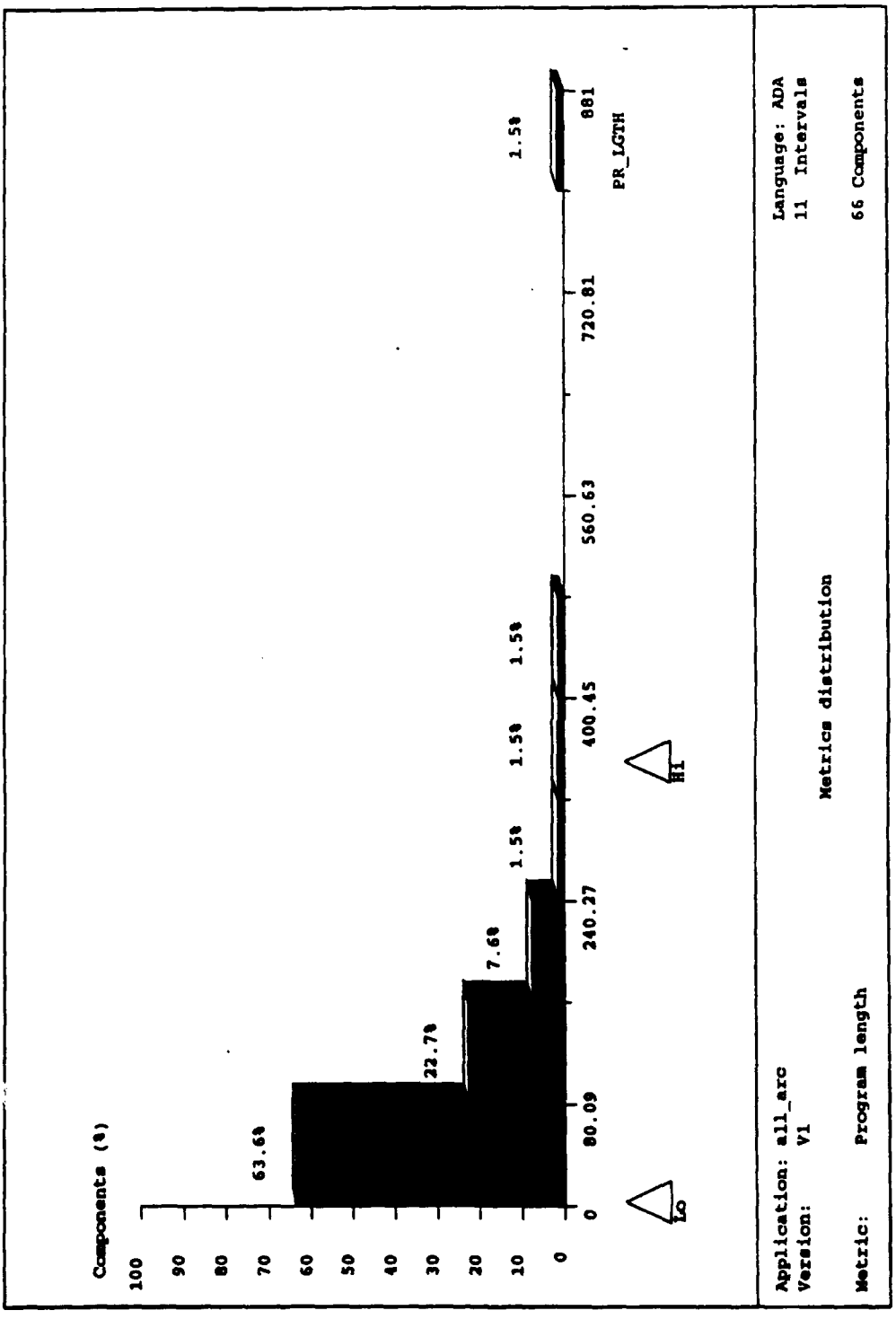


Figure 16-8. Logiscope Overall Metrics Distribution for Program Length

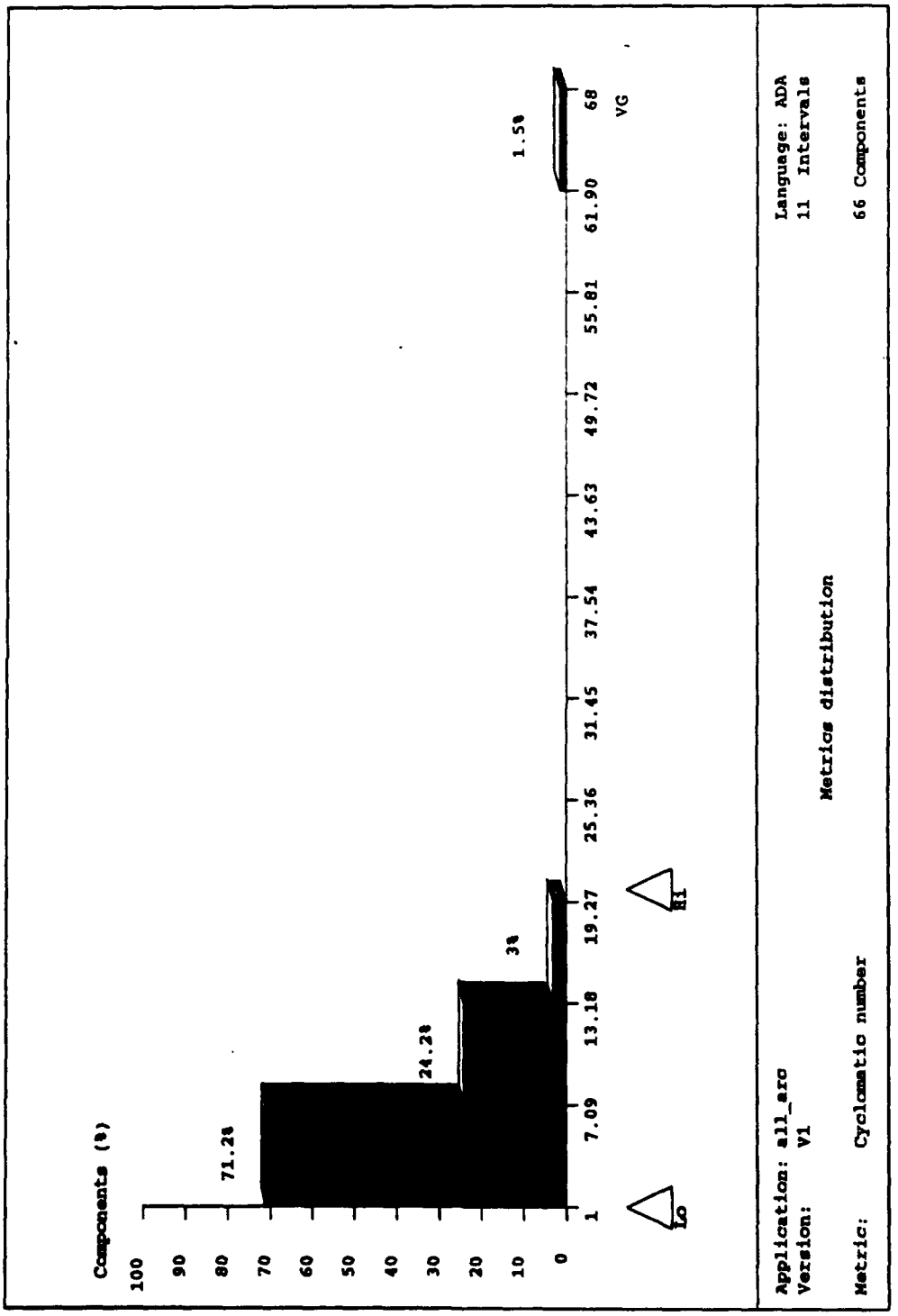


Figure 16-9. Logiscope Overall Metrics Distribution for Cyclomatic Complexity

Categories	Components	Value	%
1	LL_COMPILE/LL_TOKENS	0	
1	LL_SUPPORT	0	
			3.03
2	LL_COMPILE	1	
2	LL_SUPPORT/LOOK_AHEAD:LLATTRIBUTE:return:LLATTRIBUTE	1	
2	LL_COMPILE/LLMAIN	2	
2	LL_COMPILE/LL_TOKENS/ADVANCE/LOOK_AHEAD	3	
2	LL_SUPPORT/ALTERNATE/MERGE_RANGES:LLATTRIBUTE:LLATTRIBUTE:return:LLATTRIBUTE	3	
2	LL_SUPPORT/COMPLETE_PATTERNS	3	
2	LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE:LLSTRINGS	3	
2	LL_COMPILE/LLPRTTOKEN	4	
2	LL_SUPPORT/CONCATENATE:LLATTRIBUTE:LLATTRIBUTE:return:LLATTRIBUTE	4	
	...		
2	LL_SUPPORT/COMPLETE_PAT:LLATTRIBUTE	30	
2	LL_COMPILE/LLMAIN/READGRAM/BUILDRIGHT:INTEGER	32	
2	LL_SUPPORT/COMPLETE_PAT/COMPLETE_ALT/RESTRICT:LLATTRIBUTE:SELECTION_SET:return:LLATTRIBUTE	39	
			90.90
3	LL_SUPPORT/COMPLETE_PAT/COMPLETE_ALT/RESOLVE_AMBIGUITY:LLATTRIBUTE	56	
3	LL_SUPPORT/EMIT_SCAN_PROC	84	
3	LL_SUPPORT/EMIT_SCAN_PROC/EMIT_PATTERN_MATCH:LLATTRIBUTE:LLSTRINGS:BOOLEAN:BOOLEAN:BOOLEAN	85	
3	LL_COMPILE/LLTAKEACTION:INTEGER	85	
			6.06

List of components per metrics category

Application: all_arc
Version: V1
Language: ADA
Metric: Number of statements
Components: 66

Figure 16-10. Logiscope Components per Metrics Category for Number of Statements

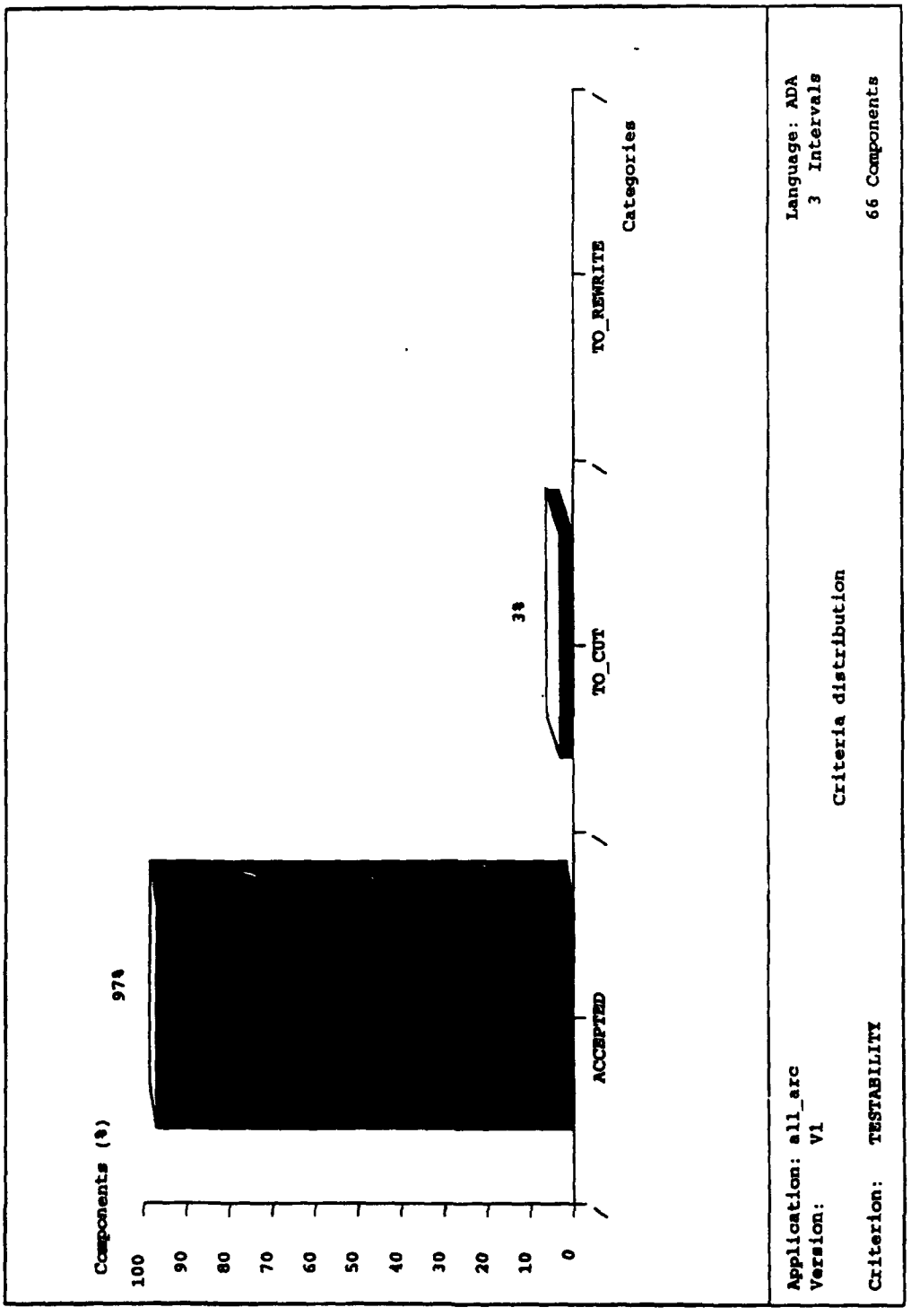


Figure 16-11. Logiscope Overall Criteria Distribution for Testability

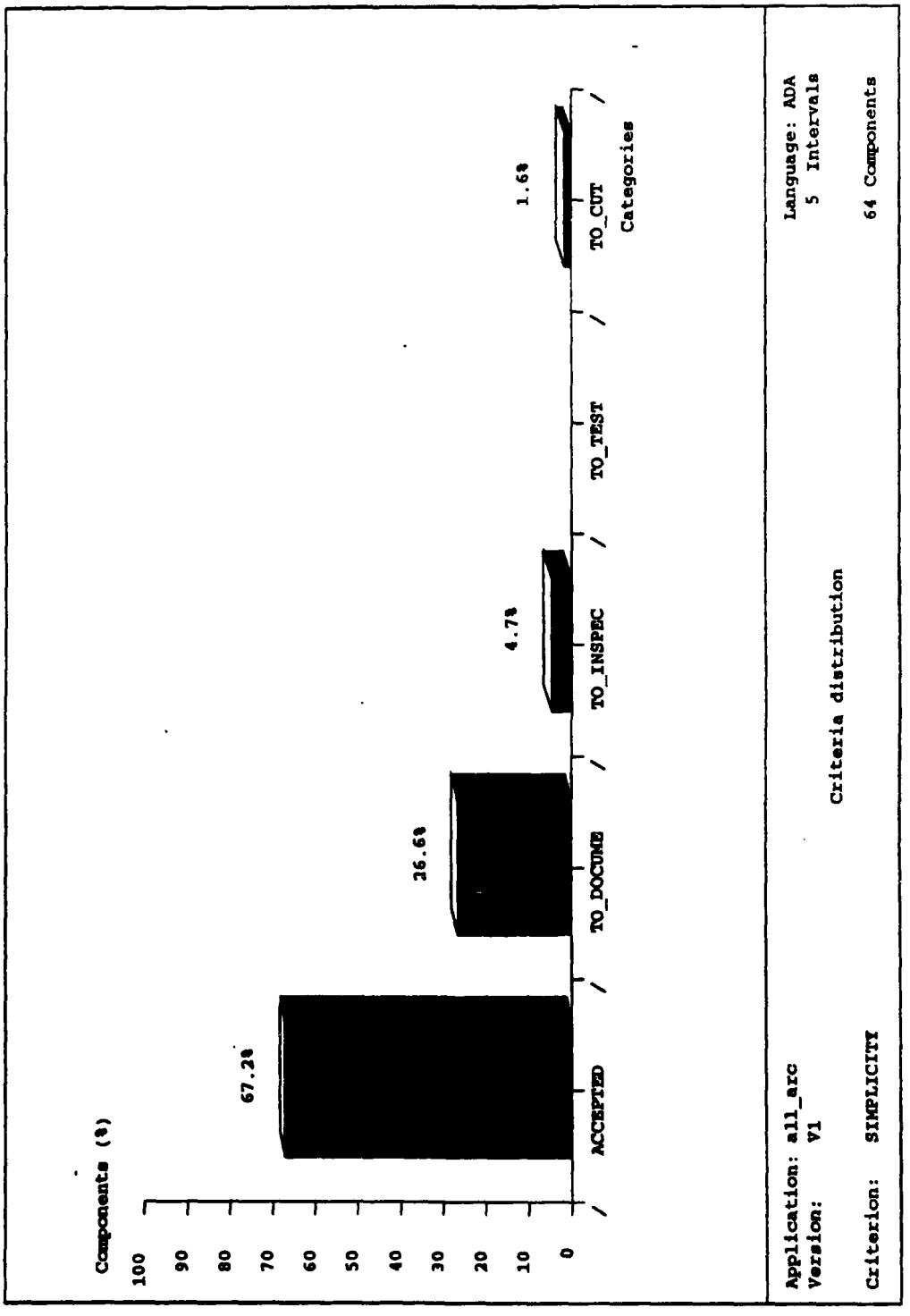


Figure 16-12. Logiscope Overall Criteria Distribution for Simplicity

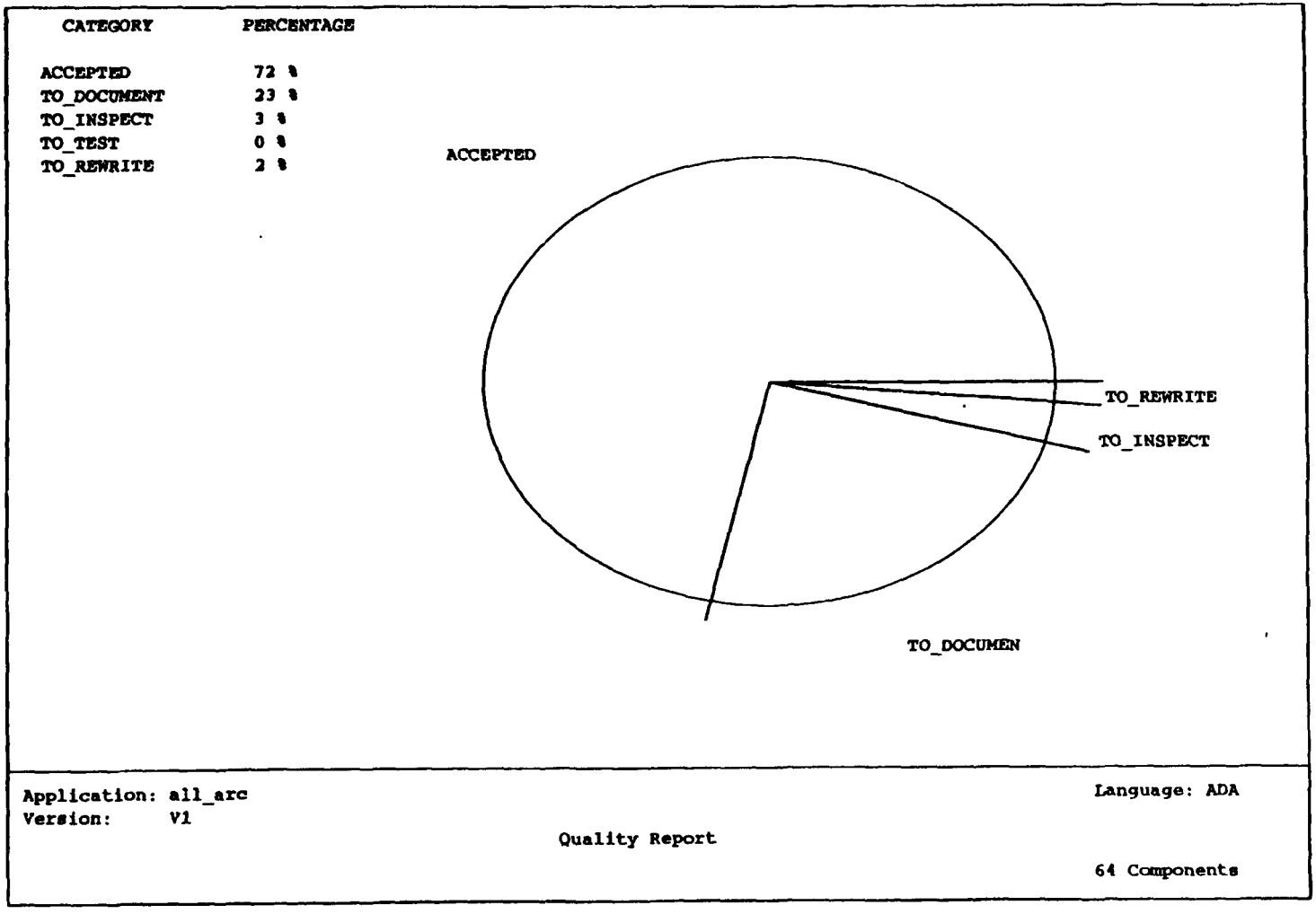


Figure 16-13. Logiscope Quality Report

```
<reference V3.1>
#  ------------------------------------------------
#  |                                              |
#  |          Quality model definition            |
#  |                                              |
#  ------------------------------------------------
                                  ...

# Quality Criteria Definition
*MC*
# The aim of this table is to explain how the components
# are classified for the TESTABILITY criterion
#-------------------------------------------------------------------------
#                          TESTABILITY                                   |
#-------------------------------------------------------------------------|
#   VG    |MAX_LVLS|  N_IO  |      associated diagnosis                   |
#-------------------------------------------------------------------------|
#   OK    |   OK   |   OK   |  ((4+4+2)/10) * 100 = 100  ACCEPTED         |
#   OK    |   OK   |        |  ((4+4+0)/10) * 100 =  80  TO_STRUCTURE     |
#   OK    |        |   OK   |  ((4+0+2)/10) * 100 =  60  TO_CUT           |
#   OK    |        |        |  ((4+0+0)/10) * 100 =  40    "              |
#         |   OK   |   OK   |  ((0+4+2)/10) * 100 =  60    "              |
#         |   OK   |        |  ((0+4+0)/10) * 100 =  40    "              |
#         |        |   OK   |  ((0+0+2)/10) * 100 =  20    "              |
#         |        |        |  ((0+0+0)/10) * 100 =   0  TO_REWRITE       |
#-------------------------------------------------------------------------

# For following languages :
# -------------------------
/ADA

# Component Level
# ---------------
# Text help definition
*AIDELECT*                              *CHAINE*
                                          Rien    = ' ADA_statement(s)'
*CONFIG*                                  Fise    = 'End Select'
    Maxdec  = 10

#  Users Metrics Definition
*MD*
       Comments Frequency          : COM_R  = N_COM/N_STMTS
       Average size of statements  : AVG_S  = PR_LGTH/N_STMTS

# Editable metrics
*ME*
  N_STMTS          I  1      50                    MAX_LVLS        I  1
  COM_R            F  0.20    1.00         N_PATHS          I  1      80
  PR_LGTH          I  3     350                 N_EXCEPT        I  1      50
  AVG_S            F  3.00    7.00         N_JUMPS          I  0       0
  VG               I  1      20
```

Figure 16-14. Logiscope Excerpt from Default Quality Model

```
# Quality Criteria Definition
*MC*
TESTABILITY =  VG + MAX_LVLS                SIMPLICITY = 2*VG + 2*N_STMTS + AVG_S
  ACCEPTED          100  100     40           ACCEPTED          100  100     30
  TO_CUT             50  100     10           TO_DOCUMENT        80  100     25
  TO_REWRITE          0   50      0           TO_INSPECT         40   80     20
                                              TO_TEST            20   40     10
                                              TO_CUT              0   20      0

LEGIBILITY = VG + PR_LGTH + MAX_LVLS + AVG_S SELF_DESCRIPTION = COM_R
  ACCEPTED          75  100     20           ACCEPTED          100  100     10
  TO_DOCUMENT       50   75     15           TO_COMMENT          0  100      0
  TO_INSPECT         0   50      0

# Quality synthesis definition
*BQ*
   ACCEPTED          90  100
   TO_DOCUMENT       80   90
   TO_INSPECT        50   80
   TO_TEST           30   50
   TO_REWRITE         0   30

#  ARCHITECTURE LEVEL
# --------------------
#  Users Metrics Definition
*AD*
   Average Paths Number/Component :  AVG_PA = CALL_PATHS/NODES

# Editable metrics
*AE*
   AVG_PA        F    1.00       2.00
   LEVELS        I    1          9
   HIER_CPX      F    1.00       5.00
   STRU_CPX      F    0.50       3.00
   ENTROPY       F    1.00       3.00

#  Quality Criteria Definition
*AC*
MODULARITY = 3*HIER_CPX + 2*STRU_CPX + 5*AVG_PA  CLARITY = ENTROPY + 2*LEVELS
   ACCEPTED         100  100    0                ACCEPTED         100   100   0
   TO_DOCUMENT       80  100    0                TO_DOCUMENT       60   100   0
   NEED_LEVELS       50   80    0                TO_PACK           30    60   0
   NEED_MODULAR       0   50    0                TO_CLARIFY         0    30   0
#  ------o-------o-------o-------o-------o-------
                                    ...
```

Figure 16-14 continued: Logiscope Excerpt from Default Quality Model

IB Test cases	1	2	3	4	Coverage
test1.lex	x	x	x	x	100.00%
sample.lex	x	x	x	x	100.00%
Total	x	x	x	x	100.00%

IB coverage of component:

LL_COMPILE/LLFIND:LLSTRINGS:LLSTYLE:return:INTEGER

Application: all_arc2
Version: V1
Language: ADA
Test Suite: TEST1.LEX

Figure 16-15. Logiscope IB Coverage of Function LLFIND

DDP / Test cases	1	2	3	4	5	6	7
IDA_Test	64	174	174	50	61	46	13
Total	64	174	174	50	61	46	13

DDP / Test cases	8	9	10	11	12	13	14
IDA_Test	4	0	79	95	144	30	0
Total	4	0	79	95	144	30	0

DDP / Test cases	15	Coverage
IDA_Test	64	86.66%
Total	64	86.66%

DDP coverage of component:

LL_COMPILE/LLMAIN/READGRAM/BUILDRIGHT:INTEGER

Application: ida
Version: V1
Language: ADA
Test Suite: CURRENT_TEST_SUITE

Figure 16-16. Logiscope DDP Coverage of Component BUILDRIGHT

DDP	Line number	Type	Condition	Executed
1	394	Begin		x
2	397	For_Loop	I in THISRHS+1..THISRHS+PRODUCTIONS(WHICHPROD).CARDRHS	x
3	398	If	I <= LLRHSSIZE	x
4	402	Case	CH = 'l'	x
5	407	Case	CH = 'a'	x
6	409	Case	CH = 'n'	x
7	414	Case	CH = 'g'	x
8	419	Case	CH = 'p'	x
9	421	Else_Case	CH <> 'l', 'a', 'n', 'g', 'p'	
10	427	If	END_OF_LINE(LLGRAM)	x
11	429	Else	not (END_OF_LINE(LLGRAM))	x
12	432	If	END_OF_LINE(LLGRAM)	x
13	434	Else	not (END_OF_LINE(LLGRAM))	x
14	438	Else	not (I <= LLRHSSIZE)	
15	446	End-For_Loop	not (I in THISRHS+1..THISRHS+PRODUCTIONS(WHICHPROD).CARDRHS)	x
DDP coverage				86.66%

DDP list of component:

LL_COMPILE/LLMAIN/READGRAM/BUILDRIGHT:INTEGER

Application: ida
Version: V1
Language: ADA
Test Suite: CURRENT_TEST_SUITE

Figure 16-16 continued: Logiscope DDP Coverage of Component BUILDRIGHT

LCSAJ Test cases	1	2	3	4	5	6	7
IDA_Test	9	0	0	9	15	5	26
Total	9	0	0	9	15	5	26

LCSAJ Test cases	8	9	10	11	12	13	14
IDA_Test	0	0	4	13	46	61	38
Total	0	0	4	13	46	61	38

LCSAJ Test cases	15	16	17	18	19	20	21
IDA_Test	12	0	79	0	65	30	144
Total	12	0	79	0	65	30	144

LCSAJ Test cases	22	23	24	25	26	27	28
IDA_Test	30	144	51	0	2	3	26
Total	30	144	51	0	2	3	26

LCSAJ Test cases	29	30	31	Coverage
IDA_Test	48	14	0	74.19%
Total	48	14	0	74.19%

```
LCSAJ coverage of component:

LL_COMPILE/LLMAIN/READGRAM/BUILDRIGHT:INTEGER

Application: ida
Version:     v1
Language:    ADA
Test Suite:   CURRENT_TEST_SUITE
```

Figure 16-17. Logiscope LCSAJ Coverage of Component BUILDRIGHT

Line number	Label	Type	Condition
		Begin	
395		*2 Statement(s)*	
397		*While*	*I in THISRHS+1..THISRHS+PRODUCTIONS(WHICHPROD).CARDRHS*
399		*If*	*I <= LLRHSSIZE*
399		*2 Statement(s)*	
401		*Case*	
403		*4 Statement(s)*	*CH = 'l'*
408		*1 Statement(s)*	*CH = 'a'*
410		*4 Statement(s)*	*CH = 'n'*
415		*4 Statement(s)*	*CH = 'g'*
420		*1 Statement(s)*	*CH = 'p'*
424		*1 Statement(s)*	*CH <> 'l', 'a', 'n', 'g', 'p'*
425		*Raise PARSING_ERROR*	
426		*End of Case*	
428		*If*	*END_OF_LINE(LLGRAM)*
428		*1 Statement(s)*	
430		*Else*	
430		*1 Statement(s)*	
431		*End If*	
433		*If*	*END_OF_LINE(LLGRAM)*
433		*1 Statement(s)*	
435		*Else*	
435		*1 Statement(s)*	
436		*End If*	
437		*1 Statement(s)*	
440		*Else*	
441		*1 Statement(s)*	
444		*Raise PARSING_ERROR*	
445		*End If*	
446		*End of While*	
447		*End*	

Type	LCSAJ numbers
Begin	1 2 3 4 5 6 7 8 . .. .
2 Statement(s)	1 2 3 4 5 6 7 8 . .. .
While	1 2 3 4 5 6 7 8 . .. . 23 24 25 26 27 28 29 30 31
If	. 2 3 4 5 6 7 8 . .. . 25 26 27 28 29 30 31
2 Statement(s)	. 2 3 4 5 6 7 . . .. 25 26 27 28 29 30 ..
Case	. 2 3 4 5 6 7 . . .. 25 26 27 28 29 30 ..
4 Statement(s)	. . . . . . 7 . . .. 30 ..
1 Statement(s)	. . . . . 6 . . . . 13 .. .. 29 .. ..
4 Statement(s)	. . . . 5 . . . . 12 .. ..

Figure 16-17 continued: Logiscope LCSAJ Coverage of Component BUILDRIGHT

```
                      .. .. .. .. .. .. .. .. 28 .. .. ..
4 Statement(s)        . . . 4 . . . . . . . . 11 .. .. .. .. .. .. .. .. ..
                      .. .. .. .. .. .. .. 27 .. .. .. ..
1 Statement(s)        . . 3 . . . . . . 10 .. .. .. .. .. .. .. .. .. ..
                      .. .. .. .. .. .. 26 .. .. .. .. .. ..
1 Statement(s)        . 2 . . . . . . 9 .. .. .. .. .. .. .. .. .. .. ..
                      .. .. .. .. .. 25 .. .. .. .. .. .. ..
Raise  PARSING_ERROR  . . . . . . . . . 9 .. .. .. .. .. .. .. .. .. .. ..
                      .. .. .. .. .. .. .. .. .. .. .. .. ..
End of Case           . . . . . . . 7 . . 10 11 12 13 14 15 .. .. .. .. ..
                      .. .. .. .. .. .. .. .. .. .. 30 ..
If                    . . . . . . . . . .. .. .. .. .. 14 15 .. .. .. .. ..
                      .. .. .. .. .. .. .. .. .. .. .. ..
1 Statement(s)        . . . . . . . . . .. .. .. .. .. 14 .. .. .. .. .. ..
                      .. .. .. .. .. .. .. .. .. .. .. ..
Else                  . . . . . . . . . .. .. .. .. .. .. 15 .. .. .. .. 19
                      20 .. .. .. .. .. .. .. .. .. .. ..
1 Statement(s)        . . . . . . . . . .. .. .. .. .. .. .. .. .. .. .. 19
                      20 .. .. .. .. .. .. .. .. .. .. ..
End If                . . . . . . . . . .. .. .. .. .. 14 .. .. 17 18 19
                      20 .. .. .. .. .. .. .. .. .. .. ..
If                    . . . . . . . . . .. .. .. .. .. .. .. .. 17 18 19
                      20 .. .. .. .. .. .. .. .. .. .. ..
1 Statement(s)        . . . . . . . . . .. .. .. .. .. .. .. .. 17 .. 19
                      .. .. .. .. .. .. .. .. .. .. .. ..
Else                  . . . . . . . . . .. .. .. .. .. .. .. .. .. 18 ..
                      20 .. 22 .. .. .. .. .. .. .. .. ..
1 Statement(s)        . . . . . . . . . .. .. .. .. .. .. .. .. .. .. ..
                      .. .. 22 .. .. .. .. .. .. .. .. ..
End If                . . . . . . . . . .. .. .. .. .. .. .. .. 17 .. 19
                      .. 21 22 .. .. .. .. .. .. .. .. ..
1 Statement(s)        . . . . . . . . . .. .. .. .. .. .. .. .. .. .. ..
                      .. 21 22 .. .. .. .. .. .. .. .. ..
Else                  . . . . . . . . 8 . .. .. .. .. .. .. 16 .. .. ..
                      .. .. .. .. .. .. .. .. .. .. .. 31
1 Statement(s)        . . . . . . . . . .. .. .. .. .. .. .. 16 .. .. ..
                      .. .. .. .. .. .. .. .. .. .. .. ..
Raise  PARSING_ERROR  . . . . . . . . . .. .. .. .. .. .. .. 16 .. .. ..
                      .. .. .. .. .. .. .. .. .. .. .. ..
End If                . . . . . . . . . .. .. .. .. .. .. .. .. .. .. ..
                      .. 21 22 23 .. .. .. .. .. .. .. ..
End of While          . . . . . . . . . .. .. .. .. .. .. .. .. .. .. ..
                      .. .. .. 23 .. .. .. .. .. .. .. ..
End                   1 . . . . . . . . .. .. .. .. .. .. .. .. .. .. ..
                      .. .. .. .. 24 .. .. .. .. .. .. ..
```

```
LCSAJ list of component:

LL_COMPILE/LLMAIN/READGRAM/BUILDRIGHT:INTEGER

Application: ida
Version:     V1
Language:    ADA
Test Suite:   CURRENT_TEST_SUITE
```

Figure 16-17 continued: Logiscope LCSAJ Coverage of Component BUILDRIGHT

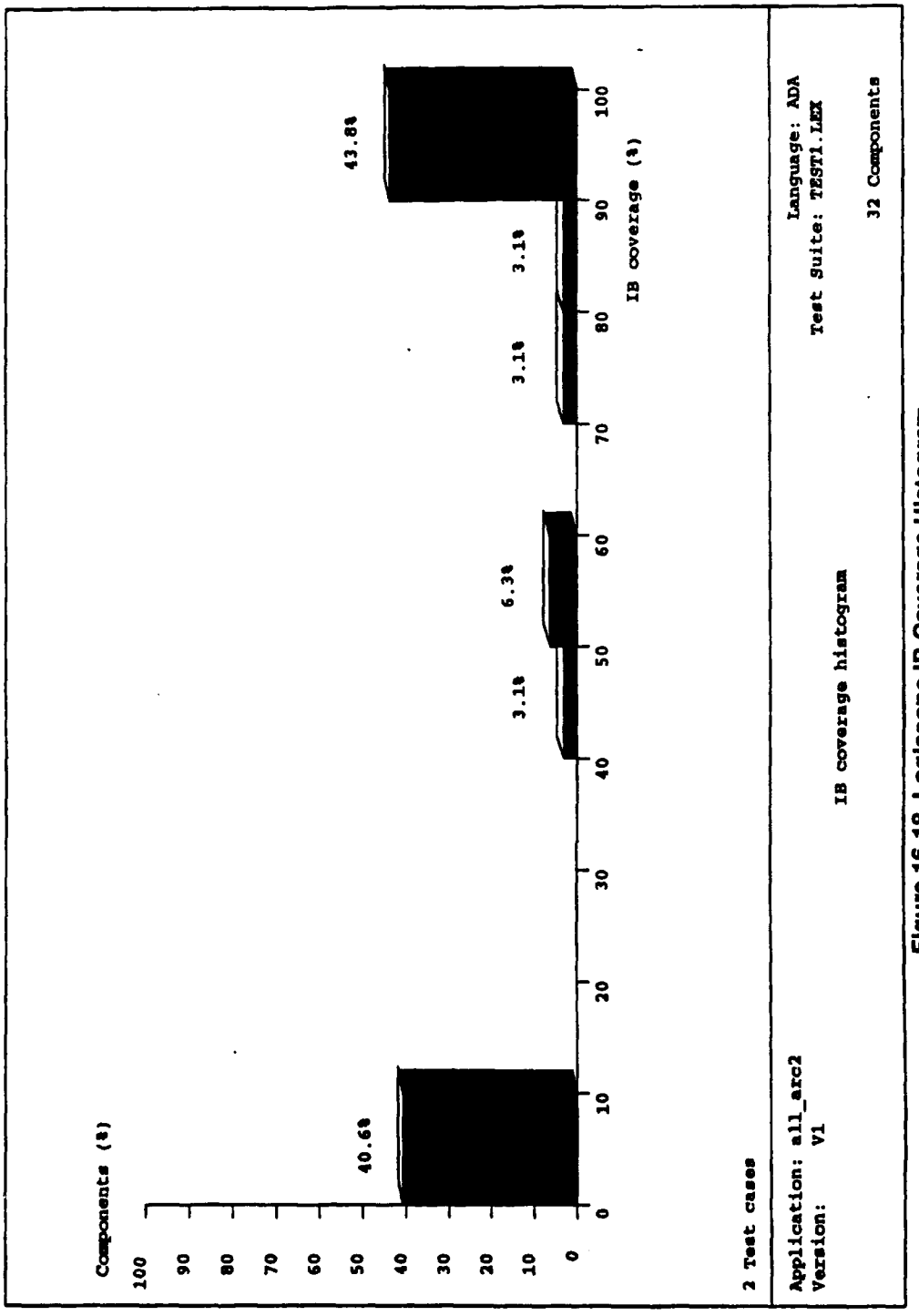


Figure 16-18. Logiscope IB Coverage Histogram

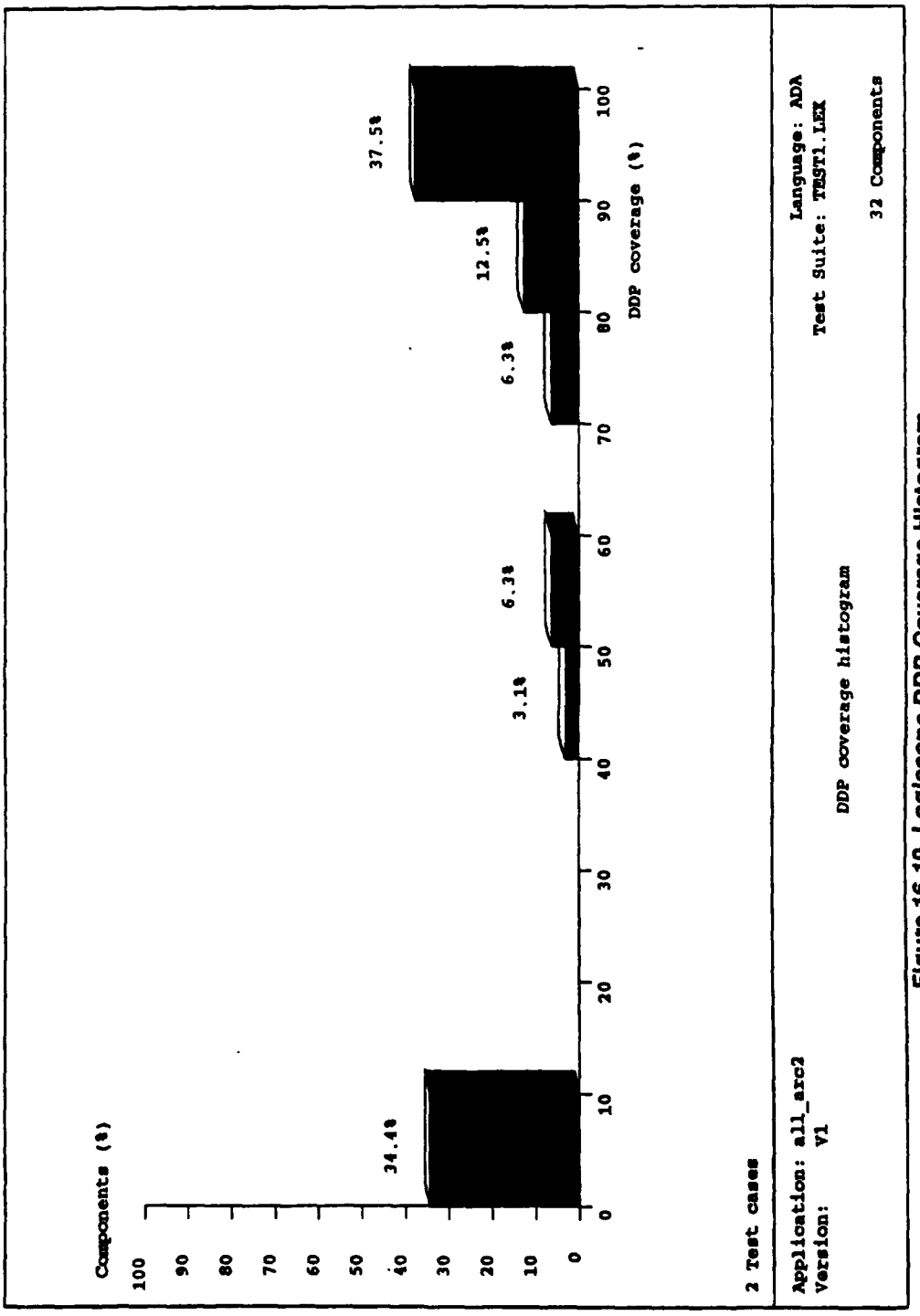


Figure 16-19. Logiscope DDP Coverage Histogram

Component	Number of IB	IB executed (%)
LL_COMPILE/LL_TOKENS	0	0.00%
LL_COMPILE/LLMAIN/PARSE/TESTSYNCH	3	0.00%
LL_COMPILE/LLMAIN/PARSE/TESTSYNCH/SYNCHRONIZE	10	0.00%
LL_COMPILE/LLMAIN/PARSE/EXPAND/MATCH:INTEGER:return:INTEGER	0	0.00%
LL_COMPILE/MAKE_TOKEN:NODE_TYPE:STRING:NATURAL:return:LLTOKEN	13	0.00%
LL_COMPILE/MAKE_TOKEN/CVT_STRING:STRING:return:LLSTRINGS	2	0.00%
LL_COMPILE/GET_CHARACTER:BOOLEAN:CHARACTER:BOOLEAN	3	0.00%
LL_COMPILE/LLFATAL	1	0.00%
LL_COMPILE/LLSKIPBOTH	1	0.00%
LL_COMPILE/LLSKIPNODE	1	0.00%
LL_COMPILE/LLSKIPTOKEN	1	0.00%
LL_COMPILE/LLPRTTOKEN	2	0.00%
LL_COMPILE/LLPRTSTRING:LLSTRINGS	3	0.00%
LL_COMPILE/LL_TOKENS/ADVANCE/NEXT_CHARACTER	5	40.00%
LL_COMPILE/LLTAKEACTION:INTEGER	68	51.47%
LL_COMPILE/LLMAIN/PARSE	12	58.33%
LL_COMPILE/LLMAIN/PARSE/EXPAND	9	77.77%
LL_COMPILE/LLMAIN/READGRAM/BUILDRIGHT:INTEGER	14	85.71%
LL_COMPILE/LLMAIN/READGRAM	9	100.00%
LL_COMPILE/LLMAIN/READGRAM/BUILDSELECT:INTEGER	3	100.00%
LL_COMPILE/LLMAIN/PARSE/ERASE	3	100.00%
LL_COMPILE/LLNEXTTOKEN	2	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/NEXT_SPEC_SYM	13	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/NEXT_IDENTIFIER	5	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE:BOOLEAN:LLTOKEN:BOOLEAN	10	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/LOOK_AHEAD	1	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/CHAR_ADVANCE	3	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/GET_CHAR:CHARACTER	3	100.00%
LL_COMPILE	1	100.00%
LL_COMPILE/LLMAIN	1	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/NEXT_STRING	4	100.00%
LL_COMPILE/LLFIND:LLSTRINGS:LLSTYLE:return:INTEGER	4	100.00%

IB coverage for test suite

Application: all_arc2
Version: V1
Language: ADA
Test Suite: TEST1.LEX

Figure 16-20. Logiscope Overall IB Coverage for Input test1.lex

Component	Number of DDP	Number of calls	DDP executed (%)
LL_COMPILE/LLPRTSTRING:LLSTRINGS	5	0	0.00%
LL_COMPILE/LLPRTTOKEN	3	0	0.00%
LL_COMPILE/LLSKIPTOKEN	1	0	0.00%
LL_COMPILE/LLSKIPNODE	1	0	0.00%
LL_COMPILE/LLSKIPBOTH	1	0	0.00%
LL_COMPILE/LLFATAL	1	0	0.00%
LL_COMPILE/GET_CHARACTER:BOOLEAN:CHARACTER:BOOLEAN	5	0	0.00%
LL_COMPILE/MAKE_TOKEN/CVT_STRING:STRING:return:LLSTRINGS	5	0	0.00%
LL_COMPILE/MAKE_TOKEN:NODE_TYPE:STRING:NATURAL:return:LLTOKEN	15	0	0.00%
LL_COMPILE/LLMAIN/PARSE/TESTSYNCH/SYNCHRONIZE	17	0	0.00%
LL_COMPILE/LLMAIN/PARSE/TESTSYNCH	5	0	0.00%
LL_COMPILE/LL_TOKENS/ADVANCE/NEXT_CHARACTER	5	31	40.00%
LL_COMPILE/LLTAKEACTION:INTEGER	69	659	52.17%
LL_COMPILE/LLMAIN/PARSE	19	2	57.89%
...			
LL_COMPILE/LL_TOKENS/ADVANCE:BOOLEAN:LLTOKEN:BOOLEAN	15	355	93.33%
LL_COMPILE	1	2	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/GET_CHAR:CHARCTER	5	2154	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/CHAR_ADVANCE	5	2152	100.00%
LL_COMPILE/LL_TOKENS/ADVANCE/LOOK_AHEAD	1	52	100.00%
LL_COMPILE/LLMAIN/READGRAM	11	2	100.00%
LL_COMPILE/LLNEXTTOKEN	3	355	100.00%
LL_COMPILE/LLFIND:LLSTRINGS:LLSTYLE:return:INTEGER	9	510	100.00%
LL_COMPILE/LLMAIN/PARSE/ERASE	5	1105	100.00%
LL_COMPILE/LLMAIN	1	2	100.00%
LL_COMPILE/LL_TOKENS	1	2	100.00%
LL_COMPILE/LLMAIN/READGRAM/BUILDSELECT:INTEGER	3	128	100.00%

```
DDP coverage for test suite

Application: all_arc2
Version:     V1
Language:    ADA
Test Suite:   TEST1.LEX
```

Figure 16-21. Logiscope Overall DDP Coverage for Input test1.lex

Metrics	Mnemonic	Value	!
Number of levels	LEVELS	2	
Hierarchy complexity	HIER_CPX	1.00	
Structural complexity	STRU_CPX	0.50	
Control entropy	ENTROPY	0.00	*
Average Paths Number/Component	AVG_PA	0.50	*

Metrics table of root:

LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE:LLSTRINGS

Application: all_arc
Version: V1
Language: ADA

Figure 16-22. Logiscope Metrics Table of Root

PATHS	Testability	Nodes
^LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE:LL STRINGS		
^^TEXT_IO/PUT:FILE_TYPE:CHARACTER	0.5000	2

Call graph path testability of root:

LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE:LLSTRINGS

Application: all_arc
Version: V1
Language: ADA

Figure 16-23. Logiscope Call Graph Path Testability of Root

Component	Access
LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE :LLSTRINGS	1.0000
TEXT_IO/PUT:FILE_TYPE:CHARACTER	1.0000

Call graph component accessibility of root:

LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE:LLSTRINGS

Application: all_arc
Version: V1
Language: ADA

Figure 16-24. Logiscope Call Graph Component Accessibility of Root

Num	Calling components	Num	Called components
49	LL_SUPPORT/EMIT_PATTERN_NAME :FILE_TYPE:LLSTRINGS	72	TEXT_IO/PUT:FILE_TYPE:CHARAC TER

Call graph calling/called components of root:

LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE:LLSTRINGS

Application: all_arc
Version: V1
Language: ADA

Figure 16-25. Logiscope Call Graph Calling/Called Components of Root

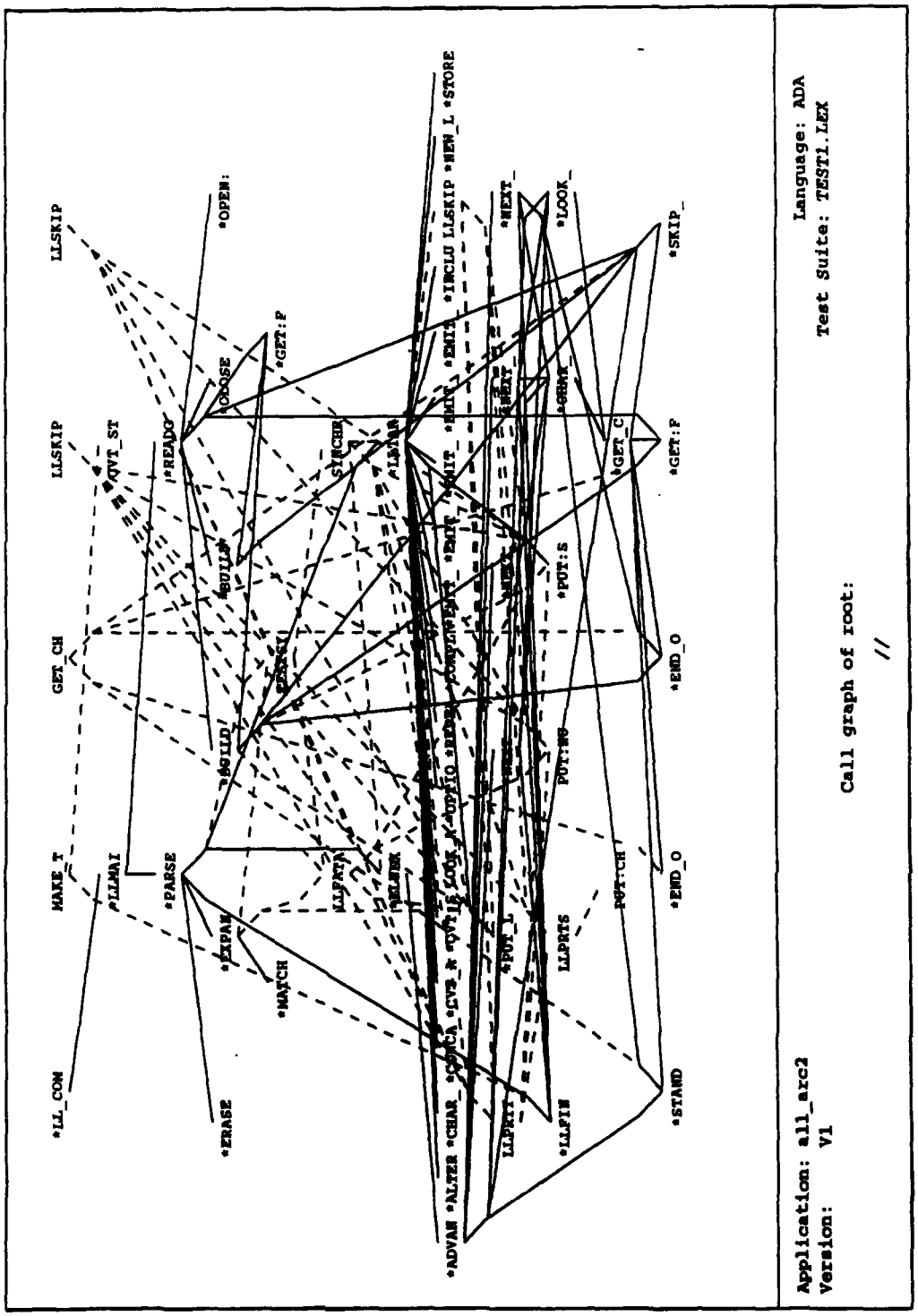


Figure 16-26. Logiscope Dynamic Call Graph of Root

Lvl	Num	List of call graph components
1	49	LL_SUPPORT/EMIT_PATTERN_NAME:FILE_TYPE:LLSTRINGS
1	22	LL_COMPILE
1	10	LL_COMPILE/MAKE_TOKEN:NODE_TYPE:STRING:NATURAL:return:LLTOKEN
1	8	LL_COMPILE/GET_CHARACTER:BOOLEAN:CHARACTER:BOOLEAN
1	6	LL_COMPILE/LLSKIPBOTH
1	4	LL_COMPILE/LLSKIPTOKEN
2	72	TEXT_IO/PUT:FILE_TYPE:CHARACTER
2	21	LL_COMPILE/LLMAIN
2	9	LL_COMPILE/MAKE_TOKEN/CVT_STRING:STRING:return:LLSTRINGS
3	20	LL_COMPILE/LLMAIN/PARSE
3	14	LL_COMPILE/LLMAIN/READGRAM
4	15	LL_COMPILE/LLMAIN/PARSE/ERASE
4	17	LL_COMPILE/LLMAIN/PARSE/EXPAND
4	12	LL_COMPILE/LLMAIN/READGRAM/BUILDRIGHT:INTEGER
4	13	LL_COMPILE/LLMAIN/READGRAM/BUILDSELECT:INTEGER
4	79	TEXT_IO/CLOSE:FILE_TYPE
4	78	TEXT_IO/OPEN:FILE_TYPE:FILE_MODE:STRING:STRING
5	16	LL_COMPILE/LLMAIN/PARSE/EXPAND/MATCH:INTEGER:return:INTEGER
5	19	LL_COMPILE/LLMAIN/PARSE/TESTSYNCH
5	80	TEXT_IO/INTEGER_IO/GET:FILE_TYPE:NUM:FIELD
6	7	LL_COMPILE/LLFATAL
6	18	LL_COMPILE/LLMAIN/PARSE/TESTSYNCH/SYNCHRONIZE
7	11	LL_COMPILE/LLNEXTTOKEN
7	66	LL_COMPILE/LLTAKEACTION:INTEGER
8	30	LL_COMPILE/LL_TOKENS/ADVANCE:BOOLEAN:LLTOKEN:BOOLEAN
		...
13	39	LL_SUPPORT/COMPLETE_PAT/COMPLETE_CONCAT:LLATTRIBUTE
13	55	LL_SUPPORT/EMIT_SCAN_PROC/EMIT_SELECT/EMIT_CHAR:CHARACTER
14	71	TEXT_IO/PUT:CHARACTER
14	68	TEXT_IO/PUT:STRING
14	33	LL_SUPPORT/ALTERNATE:LLATTRIBUTE:LLATTRIBUTE:return:LLATTRIBUTE
14	43	LL_SUPPORT/CONCATENATE:LLATTRIBUTE:LLATTRIBUTE:return:LLATTRIBUTE
15	32	LL_SUPPORT/ALTERNATE/MERGE_RANGES:LLATTRIBUTE:LLATTRIBUTE:return:LLATTRIBUTE

List of call graph components per level of root:

//
Application: all_arc
Version: V1
Language: ADA

Figure 16-27. Logiscope List of Call Graph Components per Level from Root

PPP / Test cases	1	2	3	4	5	6	7
IDA_Test	1	1	1	1	1	2	398
Total	1	1	1	1	1	2	398

PPP / Test cases	8	9	10	11	12	13	14
IDA_Test	254	0	134	230	0	254	0
Total	254	0	134	230	0	254	0

PPP / Test cases	15	16	17	18	19	20	21
IDA_Test	0	0	0	0	0	0	0
Total	0	0	0	0	0	0	0

..

PPP / Test cases	85	86	87	88	Coverage
IDA_Test	0	174	227	64	44.31%
Total	0	174	227	64	44.31%

```
Call graph PPP coverage of root:

LL_COMPILE

Application: ida
Version:     V1
Language:    ADA
Test Suite:   CURRENT_TEST_SUITE
```

Figure 16-28. Logiscope PPP Coverage of Root

17. MALPAS

MALPAS comprises a suite of static analyzers that provide control flow, data use, input/output dependency, and complexity analysis. It is unique among the examined tools in providing symbolic execution and compliance analysis of code against a formal specification.

17.1 Tool Overview

MALPAS was developed in the late 1970s at the United Kingdom Ministry of Defense Royal Signals and Radar Establishment to verify avionics and other safety-critical defense system software. Since 1986 it has been marketed and supported by TA Consultancy Services, Ltd., formerly called Rex, Thompson & Partners (RTP). MALPAS has 50 users, including 5 Ada sites. The Ada translator is a relatively new product released in July 1991. RTP also markets seminars to introduce potential customers to MALPAS and training courses. A user group is supported. MALPAS is available on VAX/VMS platforms. The tools examined in this study were MALPAS Release 5.1, IL Version 5, Pascal-IL Translator 3.1, and Ada-IL Translator 1.01. The price for MALPAS and the Ada-IL translator at the time was £60,000.

The analyses performed by MALPAS are intended to assure software safety, reliability, consistency, and conformance to standards. They include the following:

- Control flow analysis to reveal the underlying program structure and unreachable code.
- Data flow analysis to detect uninitialized variables and successive assignments without an intervening use.
- Information flow analysis to identify input-output dependencies.
- Path assessment to produce a structural complexity measure.
- Partial analysis using program slicing to reduce analysis time for semantic and compliance analysis.
- Semantic analysis to provide symbolic execution for each loop-free path.
- Compliance analysis to verify code against formal specifications.

MALPAS analyses are based on an Intermediate Language (IL) representation of program specifications or source code. Translators from several languages (including Ada, C, Fortran, and Pascal) to IL are available. The approach of using a common intermediate language for analyses simplifies the extension of MALPAS's capabilities to other program-

ming languages. Formal program specifications can also be expressed in IL. At present no automated translation tools for other formal specification languages such as OBJ, Vienna Development Method (VDM), or Z are supported.

Analyzing application source code is a two-step process. First the code is translated into IL. Since the Ada translator was not available when the tool examination started, the Pascal translator was examined first. Pascal code is translated as a single complete program; this is a straightforward process. The translation of Ada source code to IL is significantly more complicated. The sample Ada code analyzed contained several separately compiled packages and subunits. First the generic input/output packages used by the program had to be instantiated (by hand), translated, and loaded into an IL code library. Then each program unit had to be translated and loaded into the IL code library.

The second step is to run the analyses on the IL code. A single tool controls all of the available analyses. Options are selected by command line parameters and results are written to files that can be printed. Default parameter settings for initial analyses of new code were set up to include control flow, data use, and information flow analyses. Control flow, data flow, and information flow analyses are fairly standard static analysis techniques. Structured programming has largely eliminated control flow anomalies. Data flow and information flow anomalies, however, are still useful indicators of potential problems. Information flow, for example, identifies all of a subprogram's inputs and outputs, which may be more than those passed as parameters.

The compliance and semantic analyses are computationally more complex. The partial analysis capability allows these analyses to be restricted to particular modules or paths within the program. MALPAS's semantic analysis option provides symbolic execution of loop-free code segments, that is, for each possible path through a segment, the value of each modified variable is given as an algebraic expression in terms of the input variables. This provides valuable feedback to a programmer about the meaning of the code and the results that will be produced when the code is executed. The compliance testing option uses this same information to check formally specified requirements that have been added to the IL code.

17.2 Observations

Ease of use. MALPAS is a batch-oriented tool even though it may be invoked interactively. The only user interaction is through the set of options that can be selected from the

command line. The large number of options may make MALPAS "flexible" for expert users. Novice or casual users, however, may have some difficulty controlling non-default processing.

Introducing the intermediate language for analyses may cause problems for some users. All analyses and reports refer to the IL version of the program rather than to the original source code. The mapping back to the original code must be done manually. The IL approach may simplify extending MALPAS to cover a range of different programming languages (by requiring only new IL translators), but it imposes a level of separation between the actual source code and the analyses that must be compensated for by the user.

Translating Ada source code to the intermediate language was found to be somewhat more complicated than expected. The sample Ada code analyzed contained several separate packages and subunits, and normally requires several compilation steps. The MALPAS Ada to IL translator, however, required several additional steps that Ada compilers either do not need or are able to hide.

Documentation and user support. Installation and operating instructions were clear, thorough, and accurate. Installation required simply editing sample command files to name local directories and disks. The manuals included good examples and the tools operated exactly as described.

Ada restrictions. Support for all aspects of Ada that can be analyzed statically is the vendor's eventual goal, however, the current MALPAS tools support only a subset of Ada. The Ada to IL translator recognizes all valid Ada code but the translation to the intermediate language is not complete. The intermediate language, for example, does not include any mechanism for concurrency, so Ada tasks cannot be translated. This restriction is particularly unfortunate because execution-based testing of concurrent programs is often difficult to control. Repeating a particular test, for example, might not produce the same results each time. Rigorous static analyses of potential task interactions would contribute significantly to identifying and correcting tasking problems.

Translation of Ada's generic program units is not supported. Generic units provide a powerful mechanism that simplifies programs and enhances reuse. Ada's standard input and output facilities, for example, are defined in terms of generic packages. MALPAS currently requires manual instantiation of any required generic units.

Access types (pointers) and dynamic storage allocation are not supported. Analysis of unconstrained use of pointers, for example to detect potential "dangling" pointers, is virtu-

ally impossible. A workaround for disciplined use of pointers for data structures such as linked lists is to define abstract data types that encapsulate the pointers. MALPAS would be able to analyze application code that used the abstract data types since the pointers are hidden. MALPAS, however, would not be able to analyze an implementation of the abstraction that used pointers.

Problems encountered. The MALPAS tools performed as specified in their documentation. No failures occurred in use.

17.3 Planned Additions

Version V6.0 of MALPAS (due for release in November, 1992) includes two additional summary reports from the Semantic Analyser. These reports present key information from the standard Semantic Analyser report in a form that may be easier to interpret. Both reports present the conditions for and the assignments made on each path through each loop-free section of code. The Paths Table report tabulates the conditions and the assignments made to variables on each path. The Transforms report lists each variable and shows the conditions under which each assignment will be made.

17.4 Sample Outputs

Figures 17-1 through 17-6 provide sample outputs from MALPAS.

```
program  average ( input, output );
   {  This program shares a stream between two consumers by merging the  }
   {  processes and evaluating the result of the second process eagerly. }
   type    resulttype = integer;       { consumer process result type }
           streamelement = integer;    { stream element type }
   var     cons1result: resulttype;    { result returned by consumer #1 }
           cons2result: resulttype;    { result returned by consumer #2 }

   { Stream operations }
   procedure advance (var eos: Boolean; var next: streamelement; more: Boolean);
      const   CR = 13;                 { Advance the actual input stream }
      var   ch: char;
      begin
         if  more  then
            if  eof  then
               eos := true
            else begin
               eos := false;
               if  eoln  then begin
                  readln;
                  next := CR
                  end
               else begin
                  read(ch);
                  next := ord(ch)
                  end
               end
      end;

   procedure  consume;                 { Consume the input stream as one process }
      var  eos: Boolean;    { (count stream elements and sum stream elements) }
           next: streamelement;
      begin
         cons1result := 0;
         cons2result := 0;
         advance(eos,next,true);
         while  not eos  do begin
            cons1result := cons1result + 1;       { count stream elements }
            cons2result := cons2result + next;    { sum stream elements }
            advance(eos,next,true)
            end;
      end;

begin
   consume;
   writeln('The average of ', cons1result:1,
           ' characters is "', chr(cons2result div cons1result), '".')
end.
```

Figure 17-1. MALPAS Sample Pascal Code Illustrating MALPAS Analyses[1]

1. Due to MALPAS's restrictions on analysis of Ada access types, the lexical analyzer code used as a sample test program could not be thoroughly analyzed. To illustrate the reports that MALPAS produces a simple Pascal program was substituted. This program and the MALPAS analysis reports are shown in the following figures.

```
[1]             TITLE average;
[2]
[3]             [ Pascal to Malpas IL Translator - Release 3.0 ]
[4]
[6]             _INCLUDE/NOLIST "USR:[ADATEST.PASCALIL30]FIXED.PREAMBLE"
*** Including file  USR:[ADATEST.PASCALIL30]FIXED.PREAMBLE;1 ***
*** End of file  USR:[ADATEST.PASCALIL30]FIXED.PREAMBLE;1 ***
[7]             _INCLUDE/NOLIST "USR:[ADATEST.PASCALIL30]TEXT.PREAMBLE"
*** Including file  USR:[ADATEST.PASCALIL30]TEXT.PREAMBLE;1 ***
*** End of file  USR:[ADATEST.PASCALIL30]TEXT.PREAMBLE;1 ***
[8]
[10]            CONST cr : integer = +13;
[11]            CONST lit__1__theaverage : char-array = "The average of ";
[12]            CONST lit__2__characters : char-array = " characters is ";
[13]            CONST lit__3 : char-array = ".";
[14]                  [* result returned by consumer #2 *]
[16]
[17]               [* Stream operations *]
[18]
[20]            PROCSPEC advance(INOUT eos : boolean,
[21]                             INOUT next : integer,
[22]                             IN    more : boolean)
[23]            IMPLICIT ([** IL Global Parameter Section ** ]
[24]                      INOUT input : text);
*** WARNING : no DERIVES list specified for procedure  advance
[25]                  [* Advance the actual input stream *]
[26]
[27]            PROCSPEC consume
[28]            IMPLICIT ([** IL Global Parameter Section ** ]
[29]                      INOUT cons1result, cons2result : integer
[30]                      INOUT input : text);
*** WARNING : no DERIVES list specified for procedure  consume
[31]                  [* Consume the input stream as one process *]
[32]                  [* (count stream elements and sum stream elements) *]
[33]
[35]            MAINSPEC (INOUT input : text
[36]                      INOUT output : text);
[37]
[38]            PROC advance;
[40]            VAR ch: char;
[42]         #1:   IF more THEN
[43]         #3:     IF eof__text(input) THEN
[44]         #5:       eos := true
[45]                 ELSE
[46]         #6:       eos := false;
[47]         #7:       IF eoln__text(input) THEN
[48]         #9:         text__readln(input);
[49]        #10:         next := cr
[50]                 ELSE
[51]        #11:         text__read__char(input, ch);
[52]        #12:         next := char_pos(ch)
[53]         #8:       ENDIF
```

Figure 17-2. MALPAS Intermediate Language Translation of Sample

```
[54]        #4:      ENDIF
[55]        #2:    ENDIF
[56]     #STOP:   [SKIP]
[56]      #END: ENDPROC [*advance*]
[57]
[58]            PROC consume;
[60]            VAR eos__6: boolean;
[61]            VAR next__6: integer;
[63]        #1:   cons1result :=  0;
[64]        #2:   cons2result :=  0;
[65]        #3:
[65]              advance(eos__6, next__6, true);
*** WARNING : advance has not been fully specified
[66]        #4:   LOOP [while loop]
[67]        #6:   EXIT [while loop] WHEN NOT( NOT eos__6);
[68]        #7:     cons1result := cons1result +  1;
[69]                        [* count stream elements *]
[70]        #8:     cons2result := cons2result + next__6;
[71]                        [* sum stream elements *]
[72]        #9:
[72]              advance(eos__6, next__6, true)
*** WARNING : advance has not been fully specified
[73]        #5:   ENDLOOP [while loop]
[74]     #STOP:   [SKIP]
[74]      #END: ENDPROC [*consume*]
[75]
[76]            MAIN
[79]            VAR cons1result: integer;
[80]                  [* result returned by consumer #1 *]
[81]            VAR cons2result: integer;
[83]        #1:
[83]              consume();

*** WARNING : consume has not been fully specified
[84]        #2:   text__write(output, lit__1__theaverage);
                                        ...
[90]     #STOP:   [SKIP]
[90]      #END: ENDMAIN
[93]            [**** WARNING : WARNINGS IN PASS 1 ... See Listing File ]
[95]            FINISH
*** WARNING : Procedure body for text__get has not been defined
*** WARNING : Procedure body for text__page has not been defined
                                        ...
*** WARNING : Procedure body for text__writeln has not been defined
```

Figure17-2 continued: **MALPAS Intermediate Language Translation of Sample**

```
After ONE-ONE, 13 nodes removed.
No nodes with self-loops removed.
       Node id    No of pred.   Succ. nodes
       #START          0          #END
         #END          1
After KASAI (from ONE-ONE), No nodes removed.
After HECHT (from ONE-ONE), No nodes removed.
After HK (from HECHT), No nodes removed.
After TOTAL (from HK), No nodes removed.

Control Flow Summary
====================
The procedure is well structured.
The procedure has no unreachable code and no dynamic halts.
The graph was fully reduced after the following stages:
      ONE-ONE, KASAI, HECHT, HK, TOTAL
The graph was not fully reduced after the following stages:
      None
```

Figure 17-3. MALPAS Control Flow Analysis of ADVANCE

```
                                   Key
                                   ===
H = Data read and not subsequently written on some path between the nodes
I = Data read and not previously written on some path between the nodes
A = Data written twice with no intervening read on some path between the nodes
U = Data written and not subsequently read on some path between the nodes
V = Data written and not previously read on some path between the nodes
R = Data read on all paths between the nodes
W = Data written on all paths between the nodes
E = Data read on some path between the nodes
L = Data written on some path between the nodes

After ONE-ONE
   From      To     Data Use Expression
   node     node
#START    #END      H : ch   input   more
                    I : input   more
                    U : eos   input   next
                    V : ch   eos   next
                    R : more
                    E : ch   input   more
                    L : ch   eos   input   next

Summary of Possible Errors
==========================
No errors detected
```

Figure 17-4. MALPAS Data Use Analysis of ADVANCE

```
Information Flow
================

After ONE-ONE
   From node #START to node #END

      Identifier          may depend on identifier(s)

      eos                 INs/INOUTs : eos   input   more
                          CONSTANTs  : false   true
      next                INs/INOUTs : input   more   next
                          CONSTANTs  : cr
      input               INs/INOUTs : input   more
      ch                  INs/INOUTs : input   more
                          VARs/OUTs  : ch

Identifier                      may depend on conditional node(s)

eos                              #3       #1
next                             #7       #3       #1
input                            #7       #3       #1
ch                               #7       #3       #1


Summary of Possible Errors
==========================
No errors detected
```

Figure 17-5. MALPAS Information Flow Analysis of ADVANCE

```
Semantic Analysis
=================
After ONE-ONE

>From node : #START
To node   :   #END

IF NOT(more)
THEN MAP
ENDMAP
[-------------------------------------------------------------------------------]
ELSIF more AND eof__text(input)
THEN MAP
   eos := true;
ENDMAP
[-------------------------------------------------------------------------------]
ELSIF more AND eoln__text(input) AND NOT(eof__text(input))
THEN MAP
   eos := false;
   next := 13;
   input := readln__text(input);
ENDMAP
[-------------------------------------------------------------------------------]
ELSIF more AND NOT(eoln__text(input)) AND NOT(eof__text(input))
THEN MAP
   eos := false;
   next := char_pos(read__text__char(input));
   input := skip__text__char(input);
   ch := read__text__char(input);
ENDMAP ENDIF
[-------------------------------------------------------------------------------]
```

Figure 17-6. MALPAS Semantic Analysis of ADVANCE

18. QES/MANAGER

QES/Manager is one component of the QES/Workbench. To fully understand the role of QES/Manager, it is necessary first to look at the other workbench component, QES/Architect. QES/Architect is a database system designed to create and manage testing data. It has automatic capture/playback, test data generation, variable processing, and global edits. Fully instrumented testcases can automatically change or generate test data via external files, calculations, predetermined ranges, or system responses. Alternatively, test data can be imported from external sources such as screen form builders or databases, or captured from the workstation. Conditional execution is provided. By prototyping test data, usable testcases can be created that provide a picture of the user interface. These testcases can act as the specification and be used to simulate the system operation. QES/Manager is a subset of QES/Architect. (The full QES/Architect product is expected to be examined in the near future.) QES/Manager provides the data management facility that supports documenting test plans and testing activities. It also provides for easy import of ASCII data and export of data to automated test systems.

Additional workbench components expected to be released early in 1993 include QES/Qease for keystroke capture/playback, QES/Programmer for automatic unit test design and execution, and QES/Expert that aids a user in diagnosing the cause of a failure.

18.1 Tool Overview

QES/Manager is marketed by Quality Engineering Software, Inc. (QES). In addition to quality assurance (QA) products, this company markets consulting and programming services, specializing in showing customers how to improve QA and testing practices. A hotline support facility is available. QES/Manager was released in November 1991 and has over 50 users. It is language independent and runs on IBM PC/AT, or compatible machines, under MS-DOS 3.0 or higher. QES/Manager is compatible with local area networks (LANs). It supports the following test environments: DOS, 5250/AS-400 emulation, 3270 emulation, asynchronous communications, UNISYS, and Tandem. Interfaces exist to several test execution tools such as AutoTester.

The evaluation was performed on demonstration version 2.2 of QES/Manager running on a WIN TurboAT. This demonstration version is fully functional, although limited in the

number and size of testcases that can be specified. At the time of evaluation, QES/Manager prices started at $2,500.

QES/Manager embodies a predefined test model. The basic test items are as follows:

- Testcases. Define the basic unit of test data. Each Testcase is intended to be an independent, reusable testing element that tests one logical operation or module.
- Test Drivers. Group collections of Testcases so that a Test Driver consists of a list of Testcases to be run in a specified order. Further levels of grouping are available: a Test Driver List can be used to group Test Drivers, and Master Drivers to group Test Driver Lists.

Together, Testcases and Drivers form the test plan. A map function showing the developed organization of Testcases with Drivers is available from QES/Manager.

Relationships between test items are maintained using the standard nomenclature associated with databases. All named items have a keyword option which can be used for such tasks as searching and forming organizational groupings. Testcases, for example, can be classified by application or type of test, such as regression, acceptance, or system tests.

When prototyping test data, the contents of a Testcase are specified by user-defined templates. When data is imported from another test tool, QES/Manager will automatically define the necessary templates, without user intervention. In essence, a Testcase is equivalent to a template with the associated input (keystrokes) and output (responses). Access is provided to the sequence of keystrokes, not just the result. For example, once the Testcase is created, the user can view both the sequence of keystrokes and the final entry format. Different types of information can be attached to a Testcase to sequence, modify, identify, and manage it. A full-screen editor is provided for creating and modifying templates. Template fields can be either named or unnamed, however, global edits can be applied to named fields. Default responses can be specified. Instead of manually entering test data, the user can define converters that will import data from ASCII text files and transform it into QES/Manager format.

Prior to using the Drivers to guide the execution of tests, the user can view actual sequencing of test inputs and outputs to assess their correctness. In sequence mode, QES/Manager presents the operation of the application (as represented through its inputs and outputs) in its natural flow. That is, it shows the sequence of default responses, keystrokes applied to the default responses, and the response of the application to that input. A simulation function provides a similar capability, although here the user can specify time delays

to be applied to the sending of keystrokes and responses and can manually control the flow by stepping through the simulation.

The user can request reports on any test item. Essentially, the reporting facility acts as a database query engine. Here again, the user specifies templates that define a report layout in terms of cursor positions. These layouts can be saved for reuse and a copy-and-paste facility is provided, together with a sorting function.

QES/Manager provides a limited set of administration functions. These allow the system administrator to assign new users with password and access permissions, or change the access privileges of existing users. An authority matrix displays each user's rights for access to QES/Manager functions and data.

18.2 Observations

Ease of use. QES/Manager is a menu-driven system, with both mouse and keyboard navigation. Listboxes are provided to show available items for selection. QES/Manager can be customized to the extent of defining database paths, hotkeys, printer ports, and the use of color in the display screens. Conformance to IEEE and Common User Interface (CUI) standard nomenclature, interfaces, and menus is intended to facilitate use of all QES/Workbench tools.

The available on-line help includes context-sensitive help, a manual with hypertext links and a print button, and a notepad for the user to add personal help information. A tutorial and on-line demonstration is also available.

An import capability provides for importing test cases in the form of ASCII text files. These are converted into QES/Manager format using user-defined templates. A similar export capability is available.

Documentation and user support. The documentation provided with the demonstration version of the tool was fairly limited. Although it provides good guidance for stepping through one example use of the system, it does not provide a general overview of tool capabilities and usage. Installation was straightforward.

Problems encountered. QES/Manager operated as described in the documentation.

18.3 Planned Additions

OS/2 and Windows support is under development and expected to become available in the fourth quarter of 1992.

18.4 Sample Outputs

Figures 18-1 through 18-3 provide sample outputs from QES/Manager.

QES DOCUMENTATION PROJECTS

Project Name: ONLINE MANUAL — DOCUMENTATION PROJECT MANAGEMENT
Main Task: ARCHITECT MANUAL — MANUAL PROJECTS FOR ARCHITECT
Sub-task: DESIGN — DESIGN MENU DOCS

Menu Name: CAPTURE — CAPTURE Menu Documentation
New / Modify __ Entry Name:

	__How to..	__Glossary	__Index	__T.O.C.	__Main entry	__Advert/pr
done:	__	__	__	__	__	__
QA:	__	__	__	__	__	__

Menu Name: BUILD — BUILD Menu Documentation
New / Modify __ Entry Name:

	__How to..	__Glossary	__Index	__T.O.C.	__Main entry	__Advert/pr
done:	__	__	__	__	__	__
QA:	__	__	__	__	__	__

Menu Name: IMPORT — IMPORT Menu Documentation
New / Modify __ Entry Name:

	__How to..	__Glossary	__Index	__T.O.C.	__Main entry	__Advert/pr
done:	__	__	__	__	__	__
QA:	__	__	__	__	__	__

Menu Name: EXPORT — EXPORT menu documentation
New / Modify __ Entry Name:

	__How to..	__Glossary	__Index	__T.O.C.	__Main entry	__Advert/pr
done:	__	__	__	__	__	__
QA:	__	__	__	__	__	__

Menu Name:
New / Modify __ Entry Name:

	__How to..	__Glossary	__Index	__T.O.C.	__Main entry	__Advert/pr
done:	__	__	__	__	__	__
QA:	__	__	__	__	__	__

Menu Name:
New-/ Modify __ Entry Name:

	__How to..	__Glossary	__Index	__T.O.C.	__Main entry	__Advert/pr
done:	__	__	__	__	__	__
QA:	__	__	__	__	__	__

Menu Name:
New / Modify __ Entry Name:

	__How to..	__Glossary	__Index	__T.O.C.	__Main entry	__Advert/pr
done:	__	__	__	__	__	__
QA:	__	__	__	__	__	__

Figure 18-1. QES/Manager Report Layout

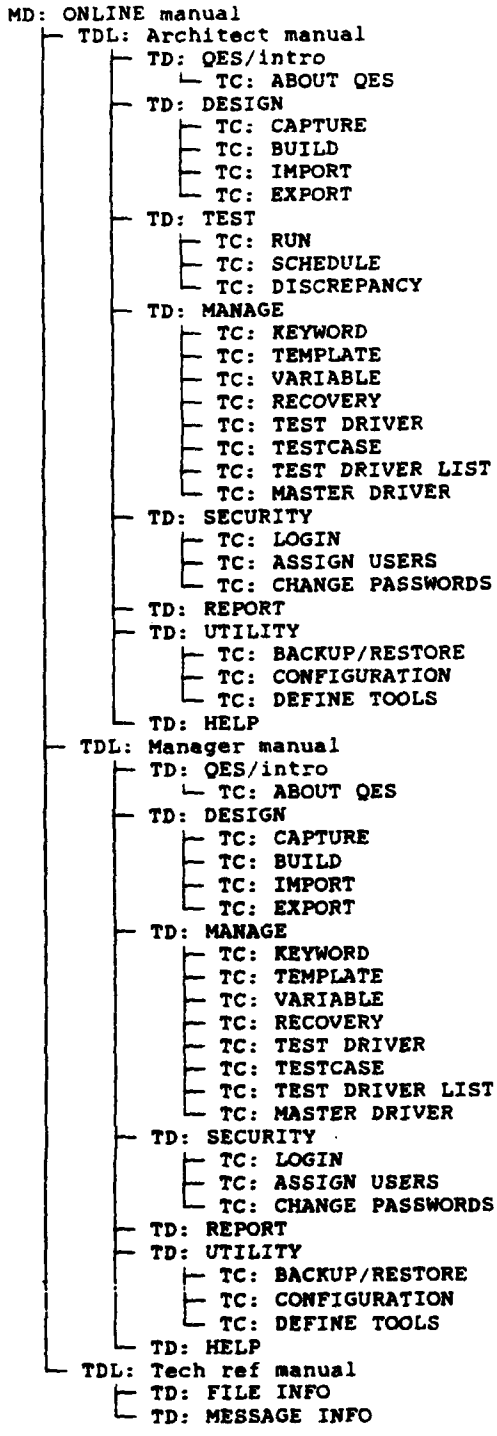


Figure 18-2. QES/Manager Map of Master Driver

PROBLEM REPORT FORM

Number/Name: 0047(F/C/03) Process Key Problem

Status:	A.Resolved	No further modifications needed
Fix/Enh:	B.Fix	Error needs correction
Severity:	C.Critical	Must be fixed in next release
Function:	D.Runable	Usable in Regression test
Product:	E.QES Architect	
Path:	P03.Testcase	

Form Generated: Tue Jul 28 14:50:57

PROBLEM RESOLUTION REPORT

Resolution Code (1-8) _____ Resolution Version _____

1 = Fixed	5 = Withdrawn by Tester
2 = Cannot Reproduce	6 = Works to Specification
3 = Fixable, but Deferred	7 = Disagree with Enhancement
4 = Cannot be Fixed	8 = Enhancement Excepted

Problem Summary _____

Problem Resolution _____

_____

_____

_____

_____

Resolved By _____ Date _____

Resolution Tested By _____ Date _____

Figure 18-3. QES/Manager Problem Report

19. SoftTest

SoftTest supports requirements-based test analysis using cause-effect graphing. It derives test conditions to guide the preparation of test data. It also provides a measure of test adequacy in terms of the number of testable functional variations for which tests have been specified.

19.1 Tool Overview

SoftTest was developed in 1987 and is marketed and supported by Bender and Associates. There are currently over 150 users. The tool runs on any IBM PC, XT, AT, PS2 or compatible platform under MS-DOS; since it executes independently of the software under test, the target or development environment of the software is not a restricting factor. Bender also markets project methodology guidelines, consulting services, and training courses on software quality assurance and testing. Interfaces via outline files of test case descriptions exist to several capture/playback tools including Automator QA, AutoTester, Gate, Microsoft Test for Windows, SQA:Robot, Sterling TestPro, V-Test, Workstation Interactive Test Tool for OS/2, XRunner, and TestRunner.

The version of SoftTest examined was Beta Release 4.0, running on a Compaq Deskpro 386/20. At the time of examination, the price was $2,500.

SoftTest automates a requirements analysis technique called cause-effect graphing, developed at IBM in the early 1970s. The primary phases of analysis are as follows:

- Extraction of node, relation, and constraint definitions from cause-effect graphs.
- Functional variation analysis to identify combinations of input conditions required for tests.
- Test condition synthesis to consolidate variations and produce minimal test sets that will exercise all the elementary functions specified.

Before using SoftTest, the user must prepare a cause-effect graph definition from the functional specification of the software under test. This process starts with identifying the set of input conditions (the causes) that a program must respond to, mapping these to the set of output conditions (the effects) that the program must produce. Unique names are assigned to each cause and to each effect, called nodes. SoftTest distinguishes primary nodes, that is, those that are basic inputs or final outputs, from intermediate nodes. In particular, SoftTest assumes that all effects that are not inputs to any other relationship (that is, prima-

ry effects) are observable and that effects that are input to other relationships are not observable unless explicitly specified as being *forced observable*. This special case of forced observable allows intermediate nodes to be used to permit testing variations where the results of one functional variation may be obscured by other variations. Relations between nodes are specified in terms of logical relations such as *and* and *or*. Finally, *exclusive*, *inclusive*, *one-and-only-one*, and *requires* constraints that restrict the invokable combinations of cause states are identified. Another statement, the *mask* statement, is included in the constraint category. It is a qualifier that is used when the true or false state of a particular node will render the state of other node(s) to be indeterminate; qualifiers work by causing specified nodes within a test case to be ignored under certain conditions.

The resulting cause-effect graph definition is expressed using a declarative, non-procedural language based on Prolog. This language includes a facility for defining a data dictionary of node names that is maintained independently of any particular cause-effect graph definition; this data dictionary can then be imported as required. A subgraph facility provides for partitioning a large specification into several parts.

Once the cause-effect graph definition has been prepared, SoftTest will perform a Variation Analysis to identify all the individual unique functions the software is required to perform. The thesis of this approach to testing is that although the number of possible combinations of input conditions may be very large, a program can be thoroughly tested by exercising this small set of unique functional variations. Some functional variations may not be testable because, for example, it may be physically impossible for certain combinations of input conditions to arise. These variations are flagged as *infeasible*. The Variation Analysis can be set to report on primitive or full-sensitized variations. In the first case, only primary nodes are included, whereas a full-sensitized analysis will include all nodes that impact a variation. Two measures of graph complexity are reported: (1) the number of functional variations divided by the number of primary causes, and (2) the number of functional variations divided by the sum of the number of primary causes and the number of primary effects. These complexity measures yield high values when inputs are combined in many different ways and low values if inputs are used in simple relationships.

The cause-effect graph definition can also be input to the Picture Presentation phase. This phase produces a pictorial representation of the cause-effect graph showing nodes, their logical relations, and any applicable constraints. It is intended to aid the user in ensuring that the cause-effect description accurately reflects his understanding of the specification's logic.

The Test Synthesis phase generates the minimum set of test cases that will ensure that all feasible functional variations are exercised. Each test case is given in the form of a test specification that identifies the causes and effects that should be true and those that should be false for this test case. (Prior to test execution, these test cases will be used to help in manually determining the actual test data needed.) Test cases can be reported in three forms. For each generated test case, a compact listing identifies the invokable causes(s) and their state(s) and the observable effect(s) and their state(s). An expanded batch listing supplements this with a full node description for each cause and effect. An expanded script listing provides much the same information but groups related causes and follows them with their associated effects for each test case. This phase also generates a fault coverage and test definition matrix that can be used for planning and tracking the test effort. The fault coverage matrix indicates the functional variations addressed by each test case and includes statistics that report on the percentage of testable variations achieved. The test definition matrix identifies the nodes included in each test case.

Test Synthesis is not restricted to the generation of new sets of test cases. For example, if a specification is changed, Test Synthesis can be used to report coverage of the revised functional variations achieved by an existing set of test cases, or to determine the new test cases that must be added to an existing set to fully cover these functional variations. As a final point, Test Synthesis itself may result in the identification of additional infeasible variations. These are variations based on any declared constraints and other relationships that are found to be indeterminate.

The latest SoftTest release includes the ability to extract test documentation. A 2167A reporting facility produces a requirements to test case traceability report that conforms to DoD-STD-2167A Section 4.1 requirements. Additionally, a Structured English requirements specification can be generated from the SoftTest input file.

19.2 Observations

Ease of use. SoftTest's user interface provides simple menu-driven commands to initiate processing, review, and print results. It is also possible to invoke one of a number of third-party text editors from within the tool so that graph specifications can be modified and analyses rerun without leaving the tool. The hard part is developing the complete cause-effect graph definition for software to be tested; this is not a limitation of the tool, but reflects the difficulty of the underlying specification task. Even though the cause-effect graph

language is clear and simple, writing specifications in this form requires some experience. Training courses offered by the vendor may also prove useful.

SoftTest can be used interactively or in batch mode. A command queue facility provides for specifying a group of cause-effect graph file name specifications to which subsequent processing will be applied.

Documentation and user support. The tool documentation and user support were quite good. Installation was simple and the tool operated exactly as described in the reference manual. Two tutorials were provided—one that worked through examples of how to run the tool and one that discussed requirements-based testing in more general terms.

Problems encountered. SoftTest performed as documented. No problems were encountered in its use.

19.3 Sample Outputs

Figures 19-1 through 19-8 provide sample outputs from SoftTest.

```
11/06/92  03:53p.m.                                  D:\CUSTCEG\IDA\LEXICON.CEG

C/E Graph Input for: C-E Graph for 4.1: Lexical Pattern Notation

TITLE   'C-E Graph for 4.1: Lexical Pattern Notation'.
/*Graphed 7-13-91  by Blaine Bragg*/
/*This graph covers page 4, section 4.1*/
/*NOTE: NT means Non_terminal expression*/
/*NOTE: RE means Regular_expression*/
/*NOTE: VB meand Vertical Bar alternative separator*/

MODES
START_ANAL='Begin the Lexical Pattern Notation Checking'.
SC_FOUND='A semi-colon was found'
        |'A semi-colon was not found'.
SC_ERROR='Display the NO SEMI-COLON FOUND error message'|/b.
DEF_LEX_END='A semi-colon was found-define as the end of the lexicon'|/b OBS.

AS_FOUND_BE='The Assign_symbol was found before the end of the lexicon'
        |'The Assign_symbol was not found'.
AS_ERROR='Diplay the NO ASSIGNMENT SYMBOL error message'|/b.
BEG_NT_EVAL='Non_terminal defined-Begin Non_terminal syntax check'|/b OBS.

CH1_EQ_LET='The first character of the Non_terminal is a letter'
        |'The first character of the Non_terminal is not a letter'.
CH1_ERROR='Display the INVALID FIRST NT CHARACTER error message'|/b.
VALID_CH1='The first character of the Non_terminal is valid' |/b OBS.

SUB_CH_VAL='The subseqent characters in the NT are valid'
        |'One or more of the subsequent NT characters are invalid'.
SUB_CHAR_EM='Display the INVALID SUBSEQUENT CHARACTER(s) error message'|/b.
DUB_US='There is a DOUBLE UNDERSCORE in the NT expression'|/b.
DUB_US_EM='Display the DOUBLE UNDRSCORE error message'|/b.
LL_LT_1LN='The NT exspression is less than or equal to one line long'
        |'The NT expression is more than one line long'.
NT_OR_EM='Display the NT EXPRESSION IS TOO LONG error message'|/b.
VALID_NT='The Non_terminal expression is valid'|/b OBS.
RE_CONT='The regular expression contains one or more characters'
        |'The regular expression has no characers (is NULL)'.
NULL_RE_EM='Display the NO REGULAR EXPRESSION error message'|/b.
BEG_RE_EVAL='Begin the REGULAR EXPRESSION syntax evaluation'|/b OBS.

QS_VALID='The Quotation Symbol syntax and contents are valid'OBS.
QS='There is one or more quotations symbols in the RE'
        |'There is no quotation sysblols in the RE'.
QS_BAL='The quotation symbols balance'.
QS_OK='The quotation symbol syntax is OK'OBS.
QS_CONT='There is one or more characters within each set of quotes'.
QS_I1=''|/b OBS.
QS_I2=''|/b OBS.
UB_QS_EM='Display the UNBALANCED QUOTATIONS error message'|/b.
NULL_QS_EM='Display the EMPTY QUOTATION SYMBOLS error message'|/b.

VB_VALID='The Vertical Bar syntax and contents are valid'|/b OBS.
VB='There is one or more VB alternative separators in the RE'|/b.

SoftTest 4.0(BETA ) ST40-000  GRAPH ENTRY Phase Input                    - 1 -
```

Figure 19-1. SoftTest Graph Entry Phase Input

```
11/06/92  03:53p.m.                                  D:\CUSTCEG\IDA\LEXICON.CEG
C/E Graph Input for: C-E Graph for 4.1: Lexical Pattern Notation

VB_LS_CONT='There is one or more characters on the left side of each VB'
        |'There are no character on the left side of each VB'.
VB_RS_CONT='There is one or more characters on the right side of each VB'.
VB_I1=''|/bOBS.
VB_I2=''|/bOBS.
VB_LS_EM='Display MISSING LEFT SIDE ALTERNATE error message'|/b.
VB_RS_EM='Display MISSING RIGHT SIDE ALTERNATE error message'|/b.

DD_VALID='The double dot syntax and contents are valid'|/b OBS.
DD='There is double dot notation in the RE'
        |'There is no double dot notation in the RE'.
DD_LS_INQ='The characters on the left side are in single quotes'
        |'The character on the left side are not in single quotes'.
DD_LS_CONT='There is one or more characters on the left side of the DD'
        |'There are no character on the left side of the DD'.
DD_RS_INQ='The characters on the right side are in single quotes'.
DD_RS_CONT='There is one or more characters on the right side of the DD'.
DD_LS_VAL='The left side of the DD is VALID'OBS.
DD_RS_VAL='The right side of the DD is VALID'OBS.
DD_LS_QEM='Display the MISSING LEFT SIDE QUOTATION error message'|/b.
DD_LS_CEM='Display the DD EMPTY LEFT SIDE QUOTATION error message'|/b.
DD_RS_QEM='Display the MISSING RIGHT SIDE QUOTATION error message'|/b.
DD_RS_CEM='Display the DD EMPTY RIGHT SIDE QUOTATION  error message'|/b.
DD_I1=''|/b OBS.
DD_I2=''|/b OBS.

BRA_VALID='The Braces syntax and contents are valid'OBS.
BRA='There are one or more braces in the RE'
        |'There are no braces in the RE'.
BRA_BAL='The Braces in the RE balance'
        |'The Braces in the RE do not balance'.
BRA_CONT='There are one or more characters within the braces'
        |'There are no character within the braces'.
BRA_I1=''|/b OBS.
BRA_I2=''|/b OBS.
BRA_UB_EM='Display the UNBALANCED BRACES error message'.
BRA_EMPTY_EM='Display the EMPTY BRACES error message'.


BRK_VALID='The Bracket syntax and contents are valid'OBS.
BRK='There are one or more Brackets in the RE'
        |'There are no Brackets in the RE'.
BRK_BAL='The Brackets in the RE balance'
        |'The brackets in the RE do not balance'.
BRK_CONT='There are one or more characters within the braces'
        |'Ther are no characters within the braces'.
BRK_I1=''|/b OBS.
BRK_I2=''|/b OBS.
BRK_UB_EM ='Display the UNBALANCED BRACKETS error message'|/b.
BRK_EMPTY_EM='Display the EMPTY BRACKETS error message'|/b.

SoftTest 4.0(BETA ) #ST40-000  GRAPH ENTRY Phase Input                    - 2 -
```

Figure 19-1 continued: SoftTest Graph Entry Phase Input

```
11/06/92  03:53p.m.                                  D:\CUSTCEG\IDA\LEXICON.CEG

C/E Graph Input for: C-E Graph for 4.1: Lexical Pattern Notation

VALID_RE='The regular expression is valid'|/b OBS.
VALID_LEX='The LEXICON IS VALID'|/b OBS.
DO_NEXT='Begin checking the next lexical statement'.



CONSTRAINTS
MASK(not BEG_RE_EVAL,QS,VB,DD,ERA,ERK).
MASK(not DEF_LEX_END,AS_FOUND_BE).
MASK(not BEG_NT_EVAL, CH1_EQ_LET).
MASK(not VALID_CH1,SUB_CH_VAL,DUB_US,LL_LT_1LN).
MASK(not VALID_NT,RE_CONT).
MASK(not QS,QS_BAL).
MASK(not QS_OK,QS_CONT).
MASK(not VB,VB_LS_CONT,VB_RS_CONT).
MASK(not DD, DD_LS_INQ,DD_LS_CONT,DD_RS_INQ,DD_RS_CONT).
MASK(not ERA,ERA_BAL,ERA_CONT).
MASK(not ERK,ERK_BAL,ERK_CONT).

RELATIONS
DEF_LEX_END:-START_ANAL and SC_FOUND.
SC_ERROR:-not SC_FOUND and START_ANAL.
BEG_NT_EVAL:-DEF_LEX_END and AS_FOUND_BE.
AS_ERROR:-DEF_LEX_END and not AS_FOUND_BE.
VALID_CH1:-BEG_NT_EVAL and CH1_EQ_LET.
CH1_ERROR:-BEG_NT_EVAL and not CH1_EQ_LET.

VALID_NT:-VALID_CH1 and SUB_CH_VAL and not DUB_US and LL_LT_1LN.
SUB_CHAR_EM:-VALID_CH1 and not SUB_CH_VAL.
DUB_US_EM:-VALID_CH1 and DUB_US.
NT_OR_EM:-VALID_CH1 AND not LL_LT_1LN.
BEG_RE_EVAL:-VALID_NT and RE_CONT.
NULL_RE_EM:-VALID_NT and not RE_CONT.

QS_OK:-QS and QS_BAL.
QS_I1:-QS_OK and QS_CONT.
QS_I2:-not QS.
QS_VALID:-QS_I1 or QS_I2.
UB_QS_EM:-QS and not QS_BAL.
NULL_QS_EM:-QS_OK and not QS_CONT.

VB_I2:-VB and VB_LS_CONT and VB_RS_CONT.
VB_I1:-not VB.
VB_LS_EM:-VB and not VB_LS_CONT.
VB_RS_EM:-VB and not VB_RS_CONT.
VB_VALID:-VB_I1 or VB_I2.

DD_LS_VAL:-DD and DD_LS_INQ and DD_LS_CONT.
DD_RS_VAL:-DD and DD_RS_INQ and DD_RS_CONT.
DD_I1:- DD_LS_VAL and DD_RS_VAL.
DD_I2:-not DD.

SoftTest 4.0(BETA ) 98740-000  GRAPH ENTRY Phase Input                - 3 -
```

Figure 19-1 continued: SoftTest Graph Entry Phase Input

```
11/06/92  03:53p.m.                                  D:\CUSTCEG\IDA\LEXICON.CEG

C/E Graph Input for: C-E Graph for 4.1: Lexical Pattern Notation

DD_VALID:-DD_I1 or DD_I2.
DD_LS_QEM:-DD and not DD_LS_INQ.
DD_LS_CEM:-DD and not DD_LS_CONT.
DD_RS_QEM:-DD and not DD_RS_INQ.
DD_RS_CEM:-DD and not DD_RS_CONT.

BRA_I1:-BRA and BRA_BAL and BRA_CONT.
BRA_I2:-not BRA.
BRA_VALID:-BRA_I1 or BRA_I2.
BRA_UB_EM:-BRA and not BRA_BAL.
BRA_EMPTY_EM:-BRA and not BRA_CONT.

BRK_I1:-BRK and BRK_BAL and BRK_CONT.
BRK_I2:-not BRK.
BRK_VALID:-BRK_I1 or BRK_I2.
BRK_UB_EM:-BRK and not BRK_BAL.
BRK_EMPTY_EM:-BRK and not BRK_CONT.

VALID_RE:-BEG_RE_EVAL and QS_VALID and VB_VALID and DD_VALID and BRA_VALID
          and BRK_VALID.
VALID_LEX:-VALID_RE.
DO_NEXT:-VALID_LEX.

TESTS

SoftTest 4.0(BETA ) #ST40-000  GRAPH ENTRY Phase Input                    - 4 -
```

Figure 19-1 continued: SoftTest Graph Entry Phase Input

```
11/06/92  04:00p.m.                                  D:\CUSTCEG\IDA\LEXICON.POV

Functional Variations: C-E Graph for 4.1: Lexical Pattern Notation

NOTE: <UNTESTABLE> and <INFEASIBLE> variations from the Test Synthesis
          phase HAVE been merged into this report.

Functional Variations for:
DEF_LEX_END:-START_ANAL AND SC_FOUND
                1. If START_ANAL and SC_FOUND
                    then DEF_LEX_END.
                2. If not START_ANAL
                     (and SC_FOUND)
                    then not DEF_LEX_END.
                3. If not SC_FOUND
                     (and START_ANAL)
                    then not DEF_LEX_END.


Functional Variations for:
SC_ERROR:-START_ANAL AND not SC_FOUND
                4. If START_ANAL and not SC_FOUND
                    then SC_ERROR.
                5. If not START_ANAL
                     (and not SC_FOUND)
                    then not SC_ERROR.
                6. If SC_FOUND
                     (and START_ANAL)
                    then not SC_ERROR.


Functional Variations for:
BEG_NT_EVAL:-DEF_LEX_END AND AS_FOUND_BE
                7. If DEF_LEX_END and AS_FOUND_BE
                    then BEG_NT_EVAL.
                8. If not DEF_LEX_END
                     (and AS_FOUND_BE MASKed)
                    then not BEG_NT_EVAL.
                9. If not AS_FOUND_BE
                     (and DEF_LEX_END)
                    then not BEG_NT_EVAL.


Functional Variations for:
AS_ERROR:-DEF_LEX_END AND not AS_FOUND_BE
               10. If DEF_LEX_END and not AS_FOUND_BE
                    then AS_ERROR.
               11. If not DEF_LEX_END
                     (and not AS_FOUND_BE MASKed)
                    then not AS_ERROR.
               12. If AS_FOUND_BE
                     (and DEF_LEX_END)
                    then not AS_ERROR.

SoftTest 4.0(BETA ) #ST40-000  VARIATION ANALYSIS Phase Output          - 1 -
```

Figure 19-2. SoftTest Variation Analysis Phase Output

```
11/06/92  04:00p.m.                                  D:\CUSTCEG\IDA\LEXICON.POV

Functional Variations: C-E Graph for 4.1: Lexical Pattern Notation

Functional Variations for:
VALID_CH1:-BEG_NT_EVAL AND CH1_EQ_LET
              13. If BEG_NT_EVAL and CH1_EQ_LET
                   then VALID_CH1.
              14. If not BEG_NT_EVAL
                    (and CH1_EQ_LET MASKed)
                   then not VALID_CH1.
              15. If not CH1_EQ_LET
                    (and BEG_NT_EVAL)
                   then not VALID_CH1.


Functional Variations for:
CH1_ERROR:-BEG_NT_EVAL AND not CH1_EQ_LET
              16. If BEG_NT_EVAL and not CH1_EQ_LET
                   then CH1_ERROR.
              17. If not BEG_NT_EVAL
                    (and not CH1_EQ_LET MASKed)
                   then not CH1_ERROR.
              18. If CH1_EQ_LET
                    (and BEG_NT_EVAL)
                   then not CH1_ERROR.


Functional Variations for:
VALID_NT:-VALID_CH1 AND SUB_CH_VAL AND LL_LT_1LN AND not DUB_US
              19. If VALID_CH1 and SUB_CH_VAL and not DUB_US and
          LL_LT_1LN
                   then VALID_NT.
              20. If not VALID_CH1
                    (and SUB_CH_VAL MASKed and not DUB_US MASKed and LL_LT_1LN
          MASKed)then not VALID_NT.
              21. If not SUB_CH_VAL
                    (and VALID_CH1 and not DUB_US and LL_LT_1LN)
                   then not VALID_NT.
              22. If not LL_LT_1LN
                    (and VALID_CH1 and SUB_CH_VAL and not DUB_US)
                   then not VALID_NT.
              23. If DUB_US
                    (and VALID_CH1 and SUB_CH_VAL and LL_LT_1LN)
                   then not VALID_NT.


Functional Variations for:
SUB_CHAR_EM:-VALID_CH1 AND not SUB_CH_VAL
              24. If VALID_CH1 and not SUB_CH_VAL
                   then SUB_CHAR_EM.
              25. If not VALID_CH1
                    (and not SUB_CH_VAL MASKed)
                   then not SUB_CHAR_EM.
              26. If SUB_CH_VAL
                                  ...

SoftTest 4.0(BETA ) #ST40-000  VARIATION ANALYSIS Phase Output              - 2 -
```

Figure 19-2 continued: SoftTest Variation Analysis Phase Output

```
11/06/92  04:00p.m.                                  D:\CUSTCEG\IDA\LEXICON.POV

Functional Variations: C-E Graph for 4.1: Lexical Pattern Notation

VALID_RE:-BEG_RE_EVAL AND QS_VALID AND VB_VALID AND DD_VALID AND BRA_VALID AND
          BRK_VALID
             129. If BEG_RE_EVAL and QS_VALID and VB_VALID and DD_VALID and
          BRA_VALID and BRK_VALID
                    then VALID_RE.
             130. If not BEG_RE_EVAL
                     (and QS_VALID and VB_VALID and DD_VALID and BRA_VALID and
          BRK_VALID)then not VALID_RE.
             131. If not QS_VALID
                     (and BEG_RE_EVAL and VB_VALID and DD_VALID and BRA_VALID
          and BRK_VALID)
                    then not VALID_RE.
             132. If not VB_VALID
                     (and BEG_RE_EVAL and QS_VALID and DD_VALID and BRA_VALID
          and BRK_VALID)
                    then not VALID_RE.
             133. If not DD_VALID
                     (and BEG_RE_EVAL and QS_VALID and VB_VALID and BRA_VALID
          and BRK_VALID)
                    then not VALID_RE.
             134. If not BRA_VALID
                     (and BEG_RE_EVAL and QS_VALID and VB_VALID and DD_VALID and
          BRK_VALID)then not VALID_RE.
             135. If not BRK_VALID
                     (and BEG_RE_EVAL and QS_VALID and VB_VALID and DD_VALID and
          BRA_VALID)then not VALID_RE.


Functional Variations for:
VALID_LEX:-VALID_RE
             136. If VALID_RE
                    then VALID_LEX.
             137. If not VALID_RE
                    then not VALID_LEX.


Functional Variations for:
DO_NEXT:-VALID_LEX
             138. If VALID_LEX
                    then DO_NEXT.
             139. If not VALID_LEX
                    then not DO_NEXT.



--> There were  NO Infeasible Variations.
--> There were  NO UNTESTABLE Variations.


SoftTest 4.0(BETA ) #5740-000  VARIATION ANALYSIS Phase Output          - 11 -
```

Figure 19-2 continued: SoftTest Variation Analysis Phase Output

```
11/06/92  04:01p.m.                                  D:\CUSTCEG\IDA\LEXICON.POT

Test Synthesis Output: C-E Graph for 4.1: Lexical Pattern Notation

[Format: NoNodeNAMES|NoRAW[Streamlined]|ALLeffects|EXTensiveTCs
['N' = Synthesis of NEW tests specified.]
['S' = Expanded-SCRIPT Test Cases requested.]

TEST CASE  1:

 Cause(s):
      Begin the Lexical Pattern Notation Checking
      A semi-colon was found
Effect(s):
      A semi-colon was found-define as the end of the lexicon

 Cause(s):
      The Assign_symbol was found before the end of the lexicon
Effect(s):
      Non_terminal defined-Begin Non_terminal syntax check

 Cause(s):
      The first character of the Non_terminal is a letter
Effect(s):
      The first character of the Non_terminal is valid

 Cause(s):
      The subseqent characters in the NT are valid
      The NT exspression is less than or equal to one line long
Effect(s):
      The Non_terminal expression is valid

 Cause(s):
      The regular expression contains one or more characters
Effect(s):
      Begin the REGULAR EXPRESSION syntax evaluation

 Cause(s):
      There is one or more quotations symbols in the RE
      The quotation symbols balance
Effect(s):
      The quotation symbol syntax is OK

 Cause(s):
      There is one or more characters within each set of quotes
Effect(s):
      The Quotation Symbol syntax and contents are valid

 Cause(s):
      There is one or more VB alternative separators in the RE
      There is one or more characters on the left side of each VB
      There is one or more characters on the right side of each VB
Effect(s):
      The Vertical Bar syntax and contents are valid


SoftTest 4.0(BETA ) #ST40-000  TEST SYNTHESIS Phase Output                - 1 -
```

Figure 19-3. SoftTest Test Synthesis Phase Output

```
11/06/92  04:01p.m.                                  D:\CUSTCEG\IDA\LEXICON.POT
Test Synthesis Output: C-E Graph for 4.1: Lexical Pattern Notation

 Cause(s):
      There is double dot notation in the RE
      The characters on the left side are in single quotes
      There is one or more characters on the left side of the DD
Effect(s):
      The left side of the DD is VALID

 Cause(s):
      The characters on the right side are in single quotes
      There is one or more characters on the right side of the DD
Effect(s):
      The right side of the DD is VALID
      The double dot syntax and contents are valid

 Cause(s):
      There are one or more braces in the RE
      The Braces in the RE balance
      There are one or more characters within the braces
Effect(s):
      The Braces syntax and contents are valid
      not Display the UNBALANCED BRACES error message
      not Display the EMPTY BRACES error message

 Cause(s):
      There are one or more Brackets in the RE
      The Brackets in the RE balance
      There are one or more characters within the braces
Effect(s):
      The Bracket syntax and contents are valid
      The regular expression is valid
      The LEXICON IS VALID
      Begin checking the next lexical statement


Source:  New test
                                        ...
-------------------------------------------------------------------------------

SoftTest 4.0(BETA ) #ST40-000  TEST SYNTHESIS Phase Output                - 2 -
```

Figure 19-3 continued: SoftTest Test Synthesis Phase Output

```
11/06/92  04:01p.m.                              D:\CUSTCEG\IDA\LEXICON.POT

Test Synthesis Output: C-E Graph for 4.1: Lexical Pattern Notation

TEST CASE 18:

 Cause(s):
      Begin the Lexical Pattern Notation Checking
      A semi-colon was found
Effect(s):
      A semi-colon was found-define as the end of the lexicon

 Cause(s):
      The Assign_symbol was found before the end of the lexicon
Effect(s):
      Non_terminal defined-Begin Non_terminal syntax check

 Cause(s):
      The first character of the Non_terminal is a letter
Effect(s):
      The first character of the Non_terminal is valid

 Cause(s):
      The subseqent characters in the NT are valid
      The NT exspression is less than or equal to one line long
Effect(s):
      The Non_terminal expression is valid

 Cause(s):
      The regular expression contains one or more characters
Effect(s):
      Begin the REGULAR EXPRESSION syntax evaluation

 Cause(s):
      There is one or more quotations symbols in the RE
      The quotation symbols balance
Effect(s):
      The quotation symbol syntax is OK

 Cause(s):
      There is one or more characters within each set of quotes
Effect(s):
      The Quotation Symbol syntax and contents are valid

 Cause(s):
      There is one or more VB alternative separators in the RE
      There is one or more characters on the left side of each VB
      There is one or more characters on the right side of each VB
Effect(s):
      The Vertical Bar syntax and contents are valid

 Cause(s):
      There is double dot notation in the RE
      The characters on the left side are in single quotes
      There is one or more characters on the left side of the DD

SoftTest 4.0(BETA ) 8ST40-000  TEST SYNTHESIS Phase Output          - 31 -
```

Figure 19-3 continued: SoftTest Test Synthesis Phase Output

```
11/06/92  04:01p.m.                                  D:\CUSTCEG\IDA\LEXICON.POT
-------------------------------------------------------------------------------
Test Synthesis Output: C-E Graph for 4.1: Lexical Pattern Notation

Effect(s):
      The left side of the DD is VALID

 Cause(s):
      The characters on the right side are in single quotes
      There is one or more characters on the right side of the DD
Effect(s):
      The right side of the DD is VALID
      The double dot syntax and contents are valid

 Cause(s):
      There are one or more braces in the RE
      The Braces in the RE balance
      There are one or more characters within the braces
Effect(s):
      The Braces syntax and contents are valid
      not Display the UNBALANCED BRACES error message
      not Display the EMPTY BRACES error message

 Cause(s):
      There are one or more Brackets in the RE
      The brackets in the RE do not balance
      There are one or more characters within the braces
Effect(s):
      not The Bracket syntax and contents are valid
      Display the UNBALANCED BRACKETS error message
      not Begin checking the next lexical statement


Source:  New test
-------------------------------------------------------------------------------
SoftTest 4.0(BETA ) #8740-000  TEST SYNTHESIS Phase Output              - 32 -
```

Figure 19-3 continued: SoftTest Test Synthesis Phase Output

11/06/92 04:01p.m. D:\CUSTCEG\IDA\LEXICON.POT

Test Synthesis Output: *C-E Graph for 4.1: Lexical Pattern Notation*

--> For n = 25 Primary Causes, then
--> 2^n = 33,554,432 THEORETICAL Maximum Number of Test Cases.

--> SoftTest generated 18 Test Cases, which yields a
--> 1,864,135 to 1 Test Case Compression Ratio.

--> SoftTest generated 139 Functional Variations, which yields a
--> 8 to 1 Functional Variations to Test Case Compression Ratio.

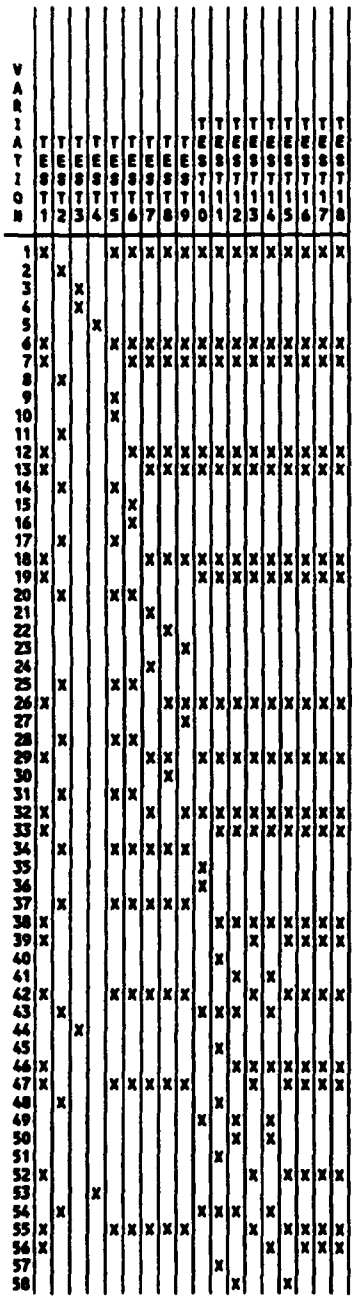
--> Test Synthesis Elapsed Time: 7 Minutes

Figure 19-3 continued: SoftTest Test Synthesis Phase Output

11/06/92 04:01p.m. D:\CUSTCBG\IDA\LEXICON.POT

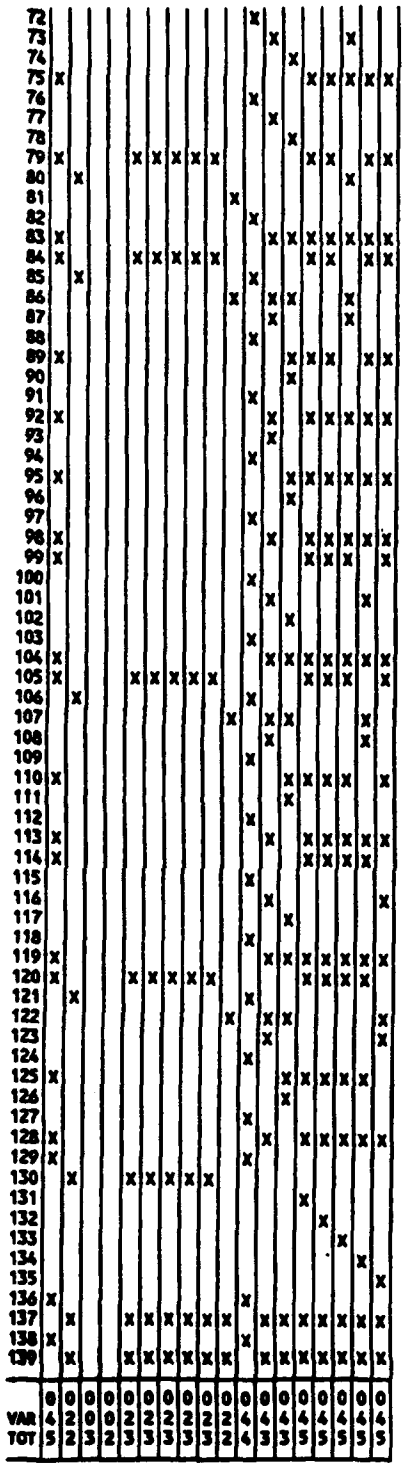
Test Synthesis Output: *C-E Graph for 4.1: Lexical Pattern Notation*

['N' = Synthesis of NEW tests specified.]
Test Case vs. Functional Variation COVERAGE MATRIX



...

Figure 19-4. SoftTest Functional Variation Coverage Matrix



SoftTest 4.0(BETA) @ST40-000 TEST SYNTHESIS Phase Output

Figure 19-4 continued: SoftTest Functional Variation Coverage Matrix

11/06/92 04:01p.m. D:\CUSTCBG\IDA\LEXICON.POT

Test Synthesis Output: *C-E Graph for 4.1: Lexical Pattern Notation*

['N' = Synthesis of NEW tests specified.]
Test Case vs. Node Name DEFINITION MATRIX

NODE-NAME	TEST 1	TEST 2	TEST 3	TEST 4	TEST 5	TEST 6	TEST 7	TEST 8	TEST 9	TEST 10	TEST 11	TEST 12	TEST 13	TEST 14	TEST 15	TEST 16	TEST 17	TEST 18
CAUSES:																		
START_ANAL	T	F	T	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
SC_FOUND	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
AS_FOUND_BE	T				F	T	T	T	T	T	T	T	T	T	T	T	T	T
CH1_EQ_LET	T					F	T	T	T	T	T	T	T	T	T	T	T	T
SUB_CH_VAL	T						F	T	T	T	T	T	T	T	T	T	T	T
DUB_US	F						F	F	T	F	F	F	F	F	F	F	F	F
LL_LT_ILN	T						T	F	T	T	T	T	T	T	T	T	T	T
RE_CONT	T									F	T	T	T	T	T	T	T	T
GS	T		T	T							F	T	T	T	T	T	T	T
GS_BAL	T		T	T	T	T	T	T	T			F	T	F	T	T	T	T
GS_CONT	T		F	F	T	T	T	T	T				T		T	T	T	T
VB	T										F	T	T	T	T	T	T	T
VB_LS_CONT	T					T	T	T	T	T			F	T	T	F	T	T
VB_RS_CONT	T					T	T	T	T	T			T	F	T	T	T	T
DD	T										F	T	T	T	T	T	T	T
DD_LS_INQ	T					T	T	T	T	T	T		F	T	T	T	F	T
DD_LS_CONT	T					T	T	T	T	T	T		T	F	T	T	T	T
DD_RS_INQ	T	T				T	T	T	T	T			F	T	T	T	T	T
DD_RS_CONT	T	T				T	T	T	T	T			T	F	T	T	T	T
BRA	T										F	T	T	T	T	T	T	T
BRA_BAL	T					T	T	T	T	T			F	T	T	T	F	T
BRA_CONT	T					T	T	T	T	T			T	F	T	T	T	T
BRK	T										F	T	T	T	T	T	T	T
BRK_BAL	T					T	T	T	T	T			F	T	T	T	T	F
BRK_CONT	T					T	T	T	T	T			T	F	T	T	T	T
EFFECTS:																		
<OBS> DEF_LEX_END	T	F	F	f	T	T	T	T	T	T	T	T	T	T	T	T	T	T
(obs) SC_ERROR	F	f	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
<OBS> BEG_NT_EVAL	T	F	f	f	F	T	T	T	T	T	T	T	T	T	T	T	T	T
(obs) AS_ERROR	F	F	f	f	T	F	F	F	F	F	F	F	F	F	F	F	F	F
<OBS> VALID_CH1	T	F	f	f	F	F	T	T	T	T	T	T	T	T	T	T	T	T
(obs) CH1_ERROR	F	F	f	f	F	T	F	F	F	F	F	F	F	F	F	F	F	F
<OBS> VALID_NT	T	F	f	f	F	F	F	F	F	T	T	T	T	T	T	T	T	T
(obs) SUB_CHAR_EN	F	F	f	f	F	F	T	F	F	F	F	F	F	F	F	F	F	F
(obs) DUB_US_EN	F	F	f	f	F	F	F	F	T	F	F	F	F	F	F	F	F	F
(obs) NT_OR_EN	F	F	f	f	F	F	F	T	F	F	F	F	F	F	F	F	F	F
<OBS> BEG_RE_EVAL	T	F	T	T	F	F	F	F	F	F	T	T	T	T	T	T	T	T
(obs) NULL_RE_EN	F	F	f	f	F	F	F	F	F	T	F	F	F	F	F	F	F	F
<OBS> GS_OK	T	F	T	T	T	T	T	T	T	F	F	T	F	T	T	T	T	T
<OBS> GS_I1	T	F	F	f	T	T	T	T	T	F	F	F	T	F	T	T	T	T
<OBS> GS_I2	F	T	f	f	F	F	F	F	F	F	T	F	F	F	F	F	F	F
<OBS> GS_VALID	T	T			T	T	T	T	T	F	T	F	T	F	T	T	T	T
(obs) UB_GS_EN	F		f	f	f	f	f	f	f		F	T	F	T	F	F	F	F
(obs) NULL_GS_EN	F	F		T	F	F	F	F	F	F	F	F	F	F	F	F	F	F
<OBS> VB_I2	T	F			T	T	T	T	T	F	F	F	T	T	F	T	T	T
<OBS> VB_I1	F	T			F	F	F	F	F	F	T	F	F	F	F	F	F	F
(obs) VB_LS_EN	F				f	f	f	f	f		F	T	F	F	T	F	F	F
(obs) VB_RS_EN	F				f	f	f	f	f		F	F	T	F	F	F	F	F
<OBS> VB_VALID	T	T			T	T	T	T	T	F	T	F	F	T	F	T	T	T
<OBS> DD_LS_VAL	T	F			T	T	T	T	T	F	F	F	T	T	T	F	T	T
<OBS> DD_RS_VAL	T	T			T	T	T	T	T	F	F	F	F	T	T	T	T	T
<OBS> DD_I1	T	F			T	T	T	T	T	F	F	F	F	T	T	F	T	T
<OBS> DD_I2	F	T			F	F	F	F	F	F	T	F	F	F	F	F	F	F
<OBS> DD_VALID	T	T			T	T	T	T	T	F	T	F	F	T	T	F	T	T
(obs) DD_LS_GEN	F				f	f	f	f	f	f	F	T	F	F	F	T	F	F
(obs) DD_LS_GEN	F				f	f	f	f	f	f	F	F	T	F	F	F	F	F

Figure 19-5. SoftTest Test Case vs. Node Name Definition Matrix

(obs)	DO_RS_GEN	F	f			f	f	f	f	f		F	T	F	F	F	F	F	F
(obs)	DO_RS_CEN	F	f			f	f	f	f	f		F	F	T	F	F	F	F	F
<OBS>	BRA_I1	T	F			T	T	T	T	T	F	F	F	F	T	T	T	F	T
<OBS>	BRA_I2	F	T			F	F	F	F	F	F	T	F	F	F	F	F	F	F
<OBS>	BRA_VALID	T	T			T	T	T	T	T	F	T	F	F	T	T	T	F	T
(obs)	BRA_UB_EN	F				f	f	f	f	f		F	T	F	F	F	F	T	F
(obs)	BRA_EMPTY_EN	F				f	f	f	f	f		F	F	T	F	F	F	F	F
<OBS>	BRK_I1	T	F			T	T	T	T	T	F	F	F	F	T	T	T	T	F
<OBS>	BRK_I2	F	T			F	F	F	F	F	F	T	F	F	F	F	F	F	F
<OBS>	BRK_VALID	T	T			T	T	T	T	T	F	T	F	F	T	T	T	T	F
(obs)	BRK_UB_EN	F				f	f	f	f	f		F	T	F	F	F	F	F	T
(obs)	BRK_EMPTY_EN	F				f	f	f	f	f		F	F	T	F	F	F	F	F
<OBS>	VALID_RE	T	F			F	F	F	F	F	F	T	F	F	F	F	F	F	F
<OBS>	VALID_LEX	T	F			F	F	F	F	F	F	T	F	F	F	F	F	F	F
(obs)	DO_NEXT	T	F			F	F	F	F	F	F	T	F	F	F	F	F	F	F

SoftTest 4.0(BETA) #ST40-000 **TEST SYNTHESIS Phase Output**

11/06/92 04:01p.m. D:\CUSTCEG\IDA\LEXICON.POT

Test Synthesis Output: ***C-E Graph for 4.1: Lexical Pattern Notation***

--> For n = 25 Primary Causes, then
--> 2^n = 33,554,432 THEORETICAL Maximum Number of Test Cases.

--> SoftTest generated 18 Test Cases, which yields a
--> 1,864,135 to 1 Test Case Compression Ratio.

--> SoftTest generated 139 Functional Variations, which yields a
--> 8 to 1 Functional Variations to Test Case Compression Ratio.

--> Test Synthesis Elapsed Time: 7 Minutes

SoftTest 4.0(BETA) #ST40-000 **TEST SYNTHESIS Phase Output**

Figure 19-5 continued: SoftTest Test Case vs. Node Name Definition Matrix

11/30/92 12:30p.m. D:\CUSTCEG\IDA\LEXICON.POP

Cause-Effect Graph for: *C-E Graph for 4.1: Lexical Pattern Notation*

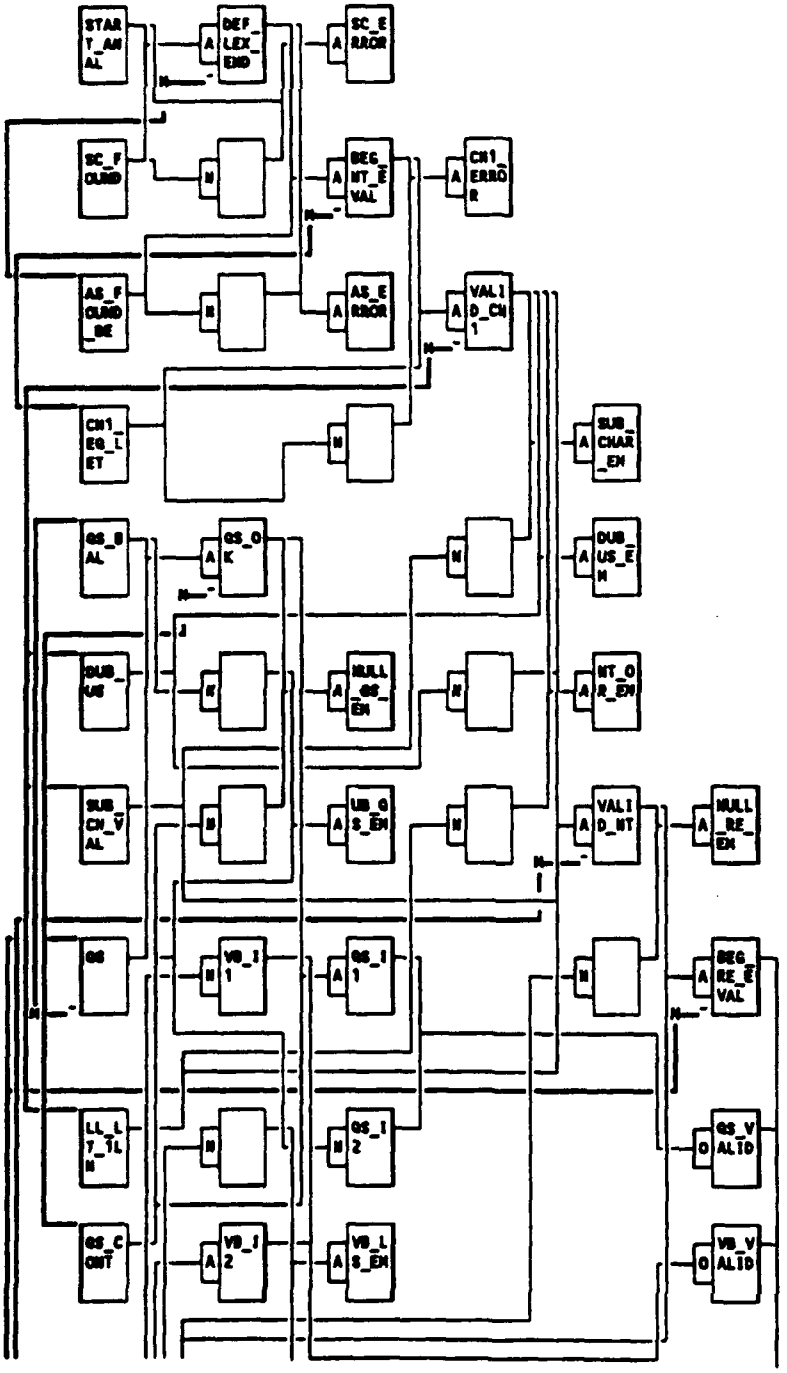


Figure 19-6. SoftTest Cause-Effect Graph

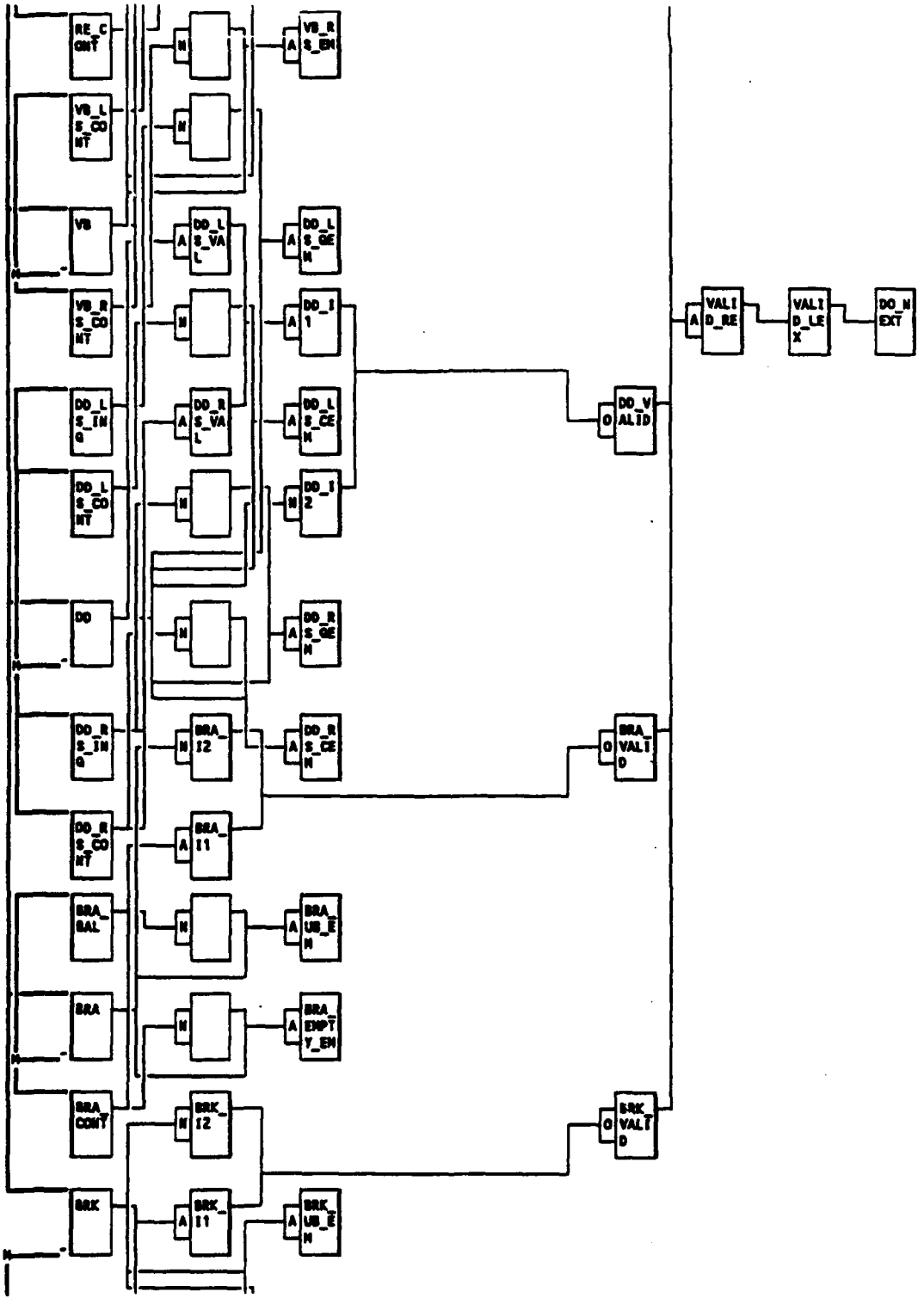
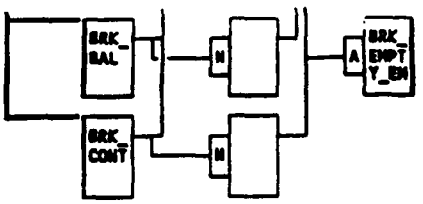


Figure 19-6 continued: SoftTest Cause-Effect Graph



SoftTest 4.0(BETA) #ST40-000 PICTURE PRESENTATION Phase Output

Figure 19-6 continued: SoftTest Cause-Effect Graph

```
11/30/92  12:12p.m.
-------------------------------------------------------------------------------
Functional Requirement Report File

Filename: D:\CUSTCEG\IDA\LEXICON.DOC

This document was extracted from SoftTest-generated data using
the following file and format specifications:
  D:\CUSTCEG\IDA\LEXICON
    Created/Last Modified: 11/06/92  04:01 p.m.
  Format: NoNodeNAMES|NoRAW[Streamlined]|ALLeffects|EXTensiveTC
  'N' = Synthesis of NEW tests specified.
  'S' = Expanded-SCRIPT Test Cases requested.
*******************************************************************************



[ NOTE: This document was created using "SoftTest":  a Computer
Assisted Software Engineering product from Bender & Associates
Inc., Larkspur, California.   SoftTest generates test case output
based on user-provided functional requirements specifications.
Based on these specifications, the following Specification Document
has been prepared.]


Functional Specifications for: C-E Graph for 4.1: Lexical Pattern Notation


  1. IF Begin the Lexical Pattern Notation Checking
        AND A semi-colon was found
      THEN A semi-colon was found-define as the end of the lexicon.

  2. IF Begin the Lexical Pattern Notation Checking
        AND A semi-colon was not found
      THEN Display the NO SEMI-COLON FOUND error message.

  3. IF A semi-colon was found-define as the end of the lexicon
        AND The Assign_symbol was found before the end of the lexicon
      THEN Non_terminal defined-Begin Non_terminal syntax check.

  4. IF A semi-colon was found-define as the end of the lexicon
        AND The Assign_symbol was not found
      THEN Diplay the NO ASSIGNMENT SYMBOL error message.

  5. IF Non_terminal defined-Begin Non_terminal syntax check
        AND The first character of the Non_terminal is a letter
      THEN The first character of the Non_terminal is valid.

  6. IF Non_terminal defined-Begin Non_terminal syntax check
        AND The first character of the Non_terminal is not a letter
      THEN Display the INVALID FIRST NT CHARACTER error message.
                                  . . .
  7. IF The first character of the Non_terminal is valid

-------------------------------------------------------------------------------
SoftTest 4.0(BETA ) #ST40-000  Functional Requirement Report File          - 1 -
```

Figure 19-7. SoftTest Functional Requirements Report

```
11/30/92  12:12p.m.
Functional Requirement Report File

 30. IF         [There are one or more Brackets in the RE
                 AND The Brackets in the RE balance
                 AND There are one or more characters within the braces]
        OR      [There are no Brackets in the RE]
       THEN The Bracket syntax and contents are valid.

 31. IF There are one or more Brackets in the RE
        AND The brackets in the RE do not balance
       THEN Display the UNBALANCED BRACKETS error message.

 32. IF There are one or more Brackets in the RE
        AND Ther are no characters within the braces
       THEN Display the EMPTY BRACKETS error message.

 33. IF Begin the REGULAR EXPRESSION syntax evaluation
        AND The Quotation Symbol syntax and contents are valid
        AND The Vertical Bar syntax and contents are valid
        AND The double dot syntax and contents are valid
        AND The Braces syntax and contents are valid
        AND The Bracket syntax and contents are valid
       THEN The regular expression is valid.

 34. IF The regular expression is valid
       THEN The LEXICON IS VALID.

 35. IF The LEXICON IS VALID
       THEN Begin checking the next lexical statement.



*******************************************************************************

In addition, the following constraints must be applied
to the above specifications:

  1. WHEN NOT Begin the REGULAR EXPRESSION syntax evaluation
       THEN the following condition(s) are INDETERMINATE:
               There is one or more quotations symbols in the RE
               There is one or more VB alternative separators in the RE
               There is double dot notation in the RE
               There are one or more braces in the RE
               There are one or more Brackets in the RE

  2. WHEN NOT A semi-colon was found-define as the end of the lexicon
       THEN the following condition(s) are INDETERMINATE:
               The Assign_symbol was found before the end of the lexicon

  3. WHEN NOT Non_terminal defined-Begin Non_terminal syntax check
       THEN the following condition(s) are INDETERMINATE:
               The first character of the Non_terminal is a letter


SoftTest 4.0(BETA ) ST40-000  Functional Requirement Report File          - 4 -
```

Figure 19-7 continued: SoftTest Functional Requirements Report

```
11/30/92  12:12p.m.

Functional Requirement Report File

  4. WHEN NOT The first character of the Non_terminal is valid
       THEN the following condition(s) are INDETERMINATE:
               The subseqent characters in the NT are valid
               There is a DOUBLE UNDERSCORE in the NT expression
               The NT exspression is less than or equal to one line long

  5. WHEN NOT The Non_terminal expression is valid
       THEN the following condition(s) are INDETERMINATE:
               The regular expression contains one or more characters

  6. WHEN There is no quotation sysblols in the RE
       THEN the following condition(s) are INDETERMINATE:
               The quotation symbols balance

  7. WHEN NOT The quotation symbol syntax is OK
       THEN the following condition(s) are INDETERMINATE:
               There is one or more characters within each set of quotes

  8. WHEN NOT There is one or more VB alternative separators in the RE
       THEN the following condition(s) are INDETERMINATE:
               There is one or more characters on the left side of each VB
               There is one or more characters on the right side of each VB

  9. WHEN There is no double dot notation in the RE
       THEN the following condition(s) are INDETERMINATE:
               The characters on the left side are in single quotes
               There is one or more characters on the left side of the DD
               The characters on the right side are in single quotes
               There is one or more characters on the right side of the DD

 10. WHEN There are no braces in the RE
       THEN the following condition(s) are INDETERMINATE:
               The Braces in the RE balance
               There are one or more characters within the braces

 11. WHEN There are no Brackets in the RE
       THEN the following condition(s) are INDETERMINATE:
               The Brackets in the RE balance
               There are one or more characters within the braces

SoftTest 4.0(BETA ) #8740-000  Functional Requirement Report File          - 5 -
```

Figure 19-7 continued: SoftTest Functional Requirements Report

```
11/30/92  12:13p.m.
-------------------------------------------------------------------------------
2167A Template Document File

Filename: D:\CUSTCEG\IDA\LEXICON.TXT

This document template was extracted from SoftTest-generated data
using the following file and format specifications:
  D:\CUSTCEG\IDA\LEXICON
    Created/Last Modified: 11/06/92  04:01 p.m.
  Format: NoNodeNAMES|NoRAW[Streamlined]|ALLeffects|EXTensiveTC
  'N' = Synthesis of NEW tests specified.
  'S' = Expanded-SCRIPT Test Cases requested.
*******************************************************************************



4.1  C-E Graph for 4.1: Lexical Pattern Notation

[ NOTE: This document was created using "SoftTest": a Computer
Assisted Software Engineering product from Bender & Associates
Inc., Larkspur, California.  SoftTest generates test case output
based on user-provided functional requirements specifications.
It is imperative that ALL of the tests specified by SoftTest be
successfully run using the SAME version of any program module(s)
under test in order to assure that full functional coverage is
achieved. ]

-------------------------------------------------------------------------------
SoftTest 4.0(BETA ) 88740-000  2167A Template Document File          - 1 -
```

Figure 19-8. SoftTest 2167A Document Template

```
11/30/92  12:13p.m.
2167A Template Document File

4.1.1  TEST CASE  LEXICON-01

This is a functional test case intended to demonstrate that the
requirements listed in the next section perform correctly.


4.1.1.1  LEXICON-01  REQUIREMENTS TRACEABILITY

This test case will test the following functional requirements:

  1. IF Begin the Lexical Pattern Notation Checking
       AND A semi-colon was found
      THEN A semi-colon was found-define as the end of the lexicon.

  2. IF Begin the Lexical Pattern Notation Checking
       AND A semi-colon was not found
      THEN Display the NO SEMI-COLON FOUND error message.

  3. IF A semi-colon was found-define as the end of the lexicon
       AND The Assign_symbol was found before the end of the lexicon
      THEN Non_terminal defined-Begin Non_terminal syntax check.

  4. IF A semi-colon was found-define as the end of the lexicon
       AND The Assign_symbol was not found
      THEN Diplay the NO ASSIGNMENT SYMBOL error message.

  5. IF Non_terminal defined-Begin Non_terminal syntax check
       AND The first character of the Non_terminal is a letter
      THEN The first character of the Non_terminal is valid.

  6. IF Non_terminal defined-Begin Non_terminal syntax check
       AND The first character of the Non_terminal is not a letter
      THEN Display the INVALID FIRST NT CHARACTER error message.

  7. IF The first character of the Non_terminal is valid
       AND The subsequent characters in the NT are valid
       AND The NT expression is less than or equal to one line long
       AND NOT There is a DOUBLE UNDERSCORE in the NT expression
      THEN The Non_terminal expression is valid.

  8. IF The first character of the Non_terminal is valid
       AND One or more of the subsequent NT characters are invalid
      THEN Display the INVALID SUBSEQUENT CHARACTER(s) error message.

  9. IF The first character of the Non_terminal is valid
       AND There is a DOUBLE UNDERSCORE in the NT expression
      THEN Display the DOUBLE UNDRSCORE error message.

 10. IF The first character of the Non_terminal is valid
       AND The NT expression is more than one line long
      THEN Display the NT EXPRESSION IS TOO LONG error message.

SoftTest 4.0(BETA ) #5740-000  2167A Template Document File          - 2 -
```

Figure 19-8 continued: SoftTest 2167A Document Template

11/30/92 12:13p.m.

2167A Template Document File

35. IF [There are one or more braces in the RE
AND The Braces in the RE balance
AND There are one or more characters within the braces]
OR [There are no braces in the RE]
THEN The Braces syntax and contents are valid.

36. IF There are one or more braces in the RE
AND The Braces in the RE do not balance
THEN Display the UNBALANCED BRACES error message.

4.1.1.2 LEXICON-01 INITIALIZATION

4.1.1.3 LEXICON-01 TEST INPUTS

1. Begin the Lexical Pattern Notation Checking
2. A semi-colon was found
3. The Assign_symbol was found before the end of the lexicon
4. The first character of the Non_terminal is a letter
5. The subsequent characters in the NT are valid
6. The NT exspression is less than or equal to one line long
7. The regular expression contains one or more characters
8. There is one or more quotations symbols in the RE
9. The quotation symbols balance
10. There is one or more characters within each set of quotes
11. There is one or more VB alternative separators in the RE
12. There is one or more characters on the left side of each VB
13. There is one or more characters on the right side of each VB
14. There is double dot notation in the RE
15. The characters on the left side are in single quotes
16. There is one or more characters on the left side of the DD
17. The characters on the right side are in single quotes
18. There is one or more characters on the right side of the DD
19. There are one or more braces in the RE
20. The Braces in the RE balance
21. There are one or more characters within the braces
22. There are one or more Brackets in the RE
23. The Brackets in the RE balance
24. There are one or more characters within the braces

4.1.1.4 LEXICON-01 EXPECTED TEST RESULTS

1. A semi-colon was found-define as the end of the lexicon
2. Non_terminal defined-Begin Non_terminal syntax check

Figure 19-8 continued: SoftTest 2167A Document Template

3. The first character of the Non_terminal is valid
4. The Non_terminal expression is valid
5. Begin the REGULAR EXPRESSION syntax evaluation
6. The quotation symbol syntax is OK
7. The Quotation Symbol syntax and contents are valid
8. The Vertical Bar syntax and contents are valid
9. The left side of the DD is VALID
10. The right side of the DD is VALID
11. The double dot syntax and contents are valid
12. The Braces syntax and contents are valid
13. NOT Display the UNBALANCED BRACES error message
14. NOT Display the EMPTY BRACES error message
15. The Bracket syntax and contents are valid
16. The regular expression is valid
17. The LEXICON IS VALID
18. Begin checking the next lexical statement

4.1.1.5 LEXICON-01 CRITERIA FOR EVALUATING RESULTS

4.1.1.6 LEXICON-01 TEST PROCEDURE

Cause(s):
Begin the Lexical Pattern Notation Checking
A semi-colon was found
Effect(s):
A semi-colon was found-define as the end of the lexicon

Cause(s):
The Assign_symbol was found before the end of the lexicon
Effect(s):
Non_terminal defined-Begin Non_terminal syntax check

Cause(s):
The first character of the Non_terminal is a letter
Effect(s):
The first character of the Non_terminal is valid

Cause(s):
The subseqent characters in the NT are valid
The NT exspression is less than or equal to one line long
Effect(s):
The Non_terminal expression is valid

Cause(s):
The regular expression contains one or more characters

Figure 19-8 continued: SoftTest 2167A Document Template

```
11/30/92  12:13p.m.
2167A Template Document File

Effect(s):
      Begin the REGULAR EXPRESSION syntax evaluation

 Cause(s):
      There is one or more quotations symbols in the RE
      The quotation symbols balance
Effect(s):
      The quotation symbol syntax is OK

 Cause(s):
      There is one or more characters within each set of quotes
Effect(s):
      The Quotation Symbol syntax and contents are valid

 Cause(s):
      There is one or more VB alternative separators in the RE
      There is one or more characters on the left side of each VB
      There is one or more characters on the right side of each VB
Effect(s):
      The Vertical Bar syntax and contents are valid

 Cause(s):
      There is double dot notation in the RE
      The characters on the left side are in single quotes
      There is one or more characters on the left side of the DD
Effect(s):
      The left side of the DD is VALID

 Cause(s):
      The characters on the right side are in single quotes
      There is one or more characters on the right side of the DD
Effect(s):
      The right side of the DD is VALID
      The double dot syntax and contents are valid

 Cause(s):
      There are one or more braces in the RE
      The Braces in the RE balance
      There are one or more characters within the braces
Effect(s):
      The Braces syntax and contents are valid
      not Display the UNBALANCED BRACES error message
      not Display the EMPTY BRACES error message

 Cause(s):
      There are one or more Brackets in the RE
      The Brackets in the RE balance
      There are one or more characters within the braces
Effect(s):
      The Bracket syntax and contents are valid
      The regular expression is valid

SoftTest 4.0(BETA ) #ST40-000  2167A Template Document File                - 7 -
```

Figure 19-8 continued: SoftTest 2167A Document Template

```
11/30/92  12:13p.m.
2167A Template Document File

      The LEXICON IS VALID
      Begin checking the next lexical statement
```


Figure 19-8 continued: SoftTest 2167A Document Template

20. SQA:Manager

SQA:Manager is a management information and decision support system for software testing. Essentially SQA:Manager provides a cataloging system for test documents and inventory items, and a tracking system to record incidents and problems found during the test process. Problem data is used to forecast quality metrics such as reliability and failure intensity. Cost data is used to report on the cost of testing and the cost of repair. The test process supported by SQA:Manager is based on a standard IEEE test methodology. As marketed, the tool can generate test reports according to the associated IEEE standards, or to conform with the appropriate U.S. Government standards. The user can, however, tailor existing formats and create new report formats.

SQA:Robot is a companion capture/playback and comparison tool for MS-Windows and OS/2 PC environments. It writes test procedures, test cases, test case results and incidents directly into the SQA:Manager database.

20.1 Tool Overview

SQA:Manager is marketed by Software Quality Automation. Based on-site needs assessment audits, this organization also develops custom testing tools. A user hot-line is available. SQA:Manager has been available since 1990, and has over 100 users. It is language independent and runs on IBM PC/AT, or compatible machines, under MS-DOS (version 3.0 or higher) with MS-Windows. It also runs under OS/2 and UNIX. A network version that operates on Novell's Netware and 3COM local area network product is available. The tool's database implementation is based on Borland International's Paradox commercial product, a relational database in conformance with IEEE recommendations. The export capabilities of this database can be used to convert SQA:Manager data for use with other PC applications such as Lotus 1-2-3. Paradox reporting functions can also be used to extend those of SQA:Manager.

The evaluation was performed on an evaluation copy of SQA:Manager version 2.0 running on a WIN TurboAT. At the time of evaluation, the price of SQA:Manager was reduced from $3,500 to $995.

SQA:Manager provides a cataloguing system for test documentation and test inventory items. It defines an entity-relationship model for the software components under test with documentation and other data generated in the testing process indexed to this model. Thus,

SQA:Manager operates on a database of test-related information specific to a software product being tested. (Multiple database are allowed so that a separate database can be set up for individual products.) The database maintains an inventory of reusable test resources, specifically test cases, test procedures, and testing tools. The specific types of test objects recognized are as follows:

- Test Plan. Details the entire testing process, that is, the items to be tested and how, the resources to be used, and the timeframe within which the testing is to occur. It provides the links between software components, test cases, and test procedures.
- Test Design Specification. Details the ways in which a software feature is to be tested and identifies the specific tests to be used.
- Test Procedure. The detailed, step-by-step instructions for test setup and operation, and evaluation of test results.
- Test Cycle. A series of tests within a release of software.
- Test Case. A test case is a set of test data designed to achieve a particular test objective. A series of test cases are run within a test cycle.
- Test Incident. An unexpected test result requiring analysis and further investigation.
- Test Log. A record over time of the details of test execution, including events for each cycle of testing. Typically, holds all activities for testing one version of given product. Test results are recorded and may reference a test incident.

These test objects can be linked to test items, that is, any software item that is to be tested, and test tools. This allows, for example, identifying which testing tools are used to run different test cases.

Incidents and problems identified in both development activities and field operation can be recorded. They are maintained separately. Problems may be entered directly into the problem database, or by classifying an incident as a problem. Problems are given a severity, resolution type, and status. They can be assigned to a specific person for resolution and so support the scheduling of resources for problem repair. Documentation of actual resolution enables accounting of related costs.

In some cases, test log, incident, or problem data may be generated by some other tool, or word processor. If this data is in the form of comma-delimited ASCII text files, templates can be created that define how it should be imported into SQA:Manager's database.

Various types of reporting are available. For a given test cycle, SQA:Manager produces reports identifying what test cases were run and their results, and the summarized results of all test cases. For a given test case, the results through all test cycles can be reported. Cost reports are available for the cost of testing and the cost of repair. General reporting on

problems and incidents experienced is available. Problem data is also used to generate reliability metrics using Musa's logarithmic Poisson execution time model. Failure intensity reports indicate the expected number of failures per unit of time, whereas reliability reports give the probability of failure-free operation. Both types of report provide additional information such as the amount of additional testing time needed to meet a targeted reliability objective. The accuracy of these estimates depends on the amount of problem data. SQA:-Manager documentation recommends that the reliability model is only applied if there are more than 100 problem reports, testers are at least 25% through testing, and the software under test exceeds 25,000 lines of code. Finally, reports on the software under test, documents, inventory contents, and test log are available.

Many of the reports are available as either tables, bar charts, or line graphs. Before printing any predefined report, the user can change the report title and *x*- and *y*-axis labels. Some dozen filters are predefined for both incident and problems reports. These can be used, for example, to limit problem reporting to problems not yet resolved. The user can easily prepare templates that generate additional reports, defining the items to appear along with primary and secondary sorting keys.

Predefined templates are provided for test plans and test design specifications. SQA:-Manager comes with three prepackaged template groups, as follows:

- IEEE Standards
  a. IEEE Std 730, Software Quality Assurance Plans
  b. IEEE Std 829, Software Test Documentation
  c. IEEE Std 1008, Software Unit Testing
  d. IEEE Std 1028, Software Reviews and Audits
- U.S. Military Standards
  a. MIL-STD-480, Configuration Management
- U.S. Department of Defense Data Item Descriptions and Standards
  a. DI-MCCR-80014, Software Test Plans
  b. DI-MCCR-80015, Software Test Description
  c. DI-MCCR-80017, Software Test Report
  d. DI-QCIC-80572, Software Quality Program Plan
  e. DoD-STD-2167A, (Excerpt) Defense System Software Development

SQA:Manager includes a set of administration functions. These allow an administrator to specify the relationships between a product, programs (a logical division of products),

modules (a logical division of programs), documents, and versions. The ability to reference a program, or module, from more than one product supports software reuse.

Security is provided by identifying allowable users, each of whom is given a password. Users are distinguished by organization to allow, for example, identifying problems reported by customers as opposed to those reported by testers. Each user is given a particular set of access privileges. In the case of problem reporting, for example, a user can be given the ability to add or change descriptions, set the status, enter resolution or follow-up action and, for a problem, assign the person responsible for resolution. Setting up for a new project requires establishing a cost base that will be used for cost reporting. Here the administrator can define an unlimited number of cost codes, each of which has an associated hourly rate. Finally, the administrator can establish the necessary operating profile, such as the phone numbers and communications ports to be used with remote communications.

20.2 Observations

Ease of use. SQA:Manager provides a menu-driven interface, where the user may use either the keyboard or mouse to make selections. Three types of screens are supported: field entry screens, check boxes, and item selector lists. After a basic function has been selected, the tool guides a user through the steps in that function, capturing information through templates. Where appropriate, a pick list function is provided to display the options for a text field and allow the user to select from this list. Graphical outputs are available in the form of bar charts, line graphs, and tables. Context-sensitive on-line help is supported, but provides only terse messages.

A programming module is provided to facilitate customizing or extending SQA:Manager. The tool can be customized in several ways. The text in reports and help messages can be changed, along with field and button labels. The contents of field entry screens and item selector lists can be modified. The document templates can be changed, for example, to reflect particular organizational or product requirements. As previously mentioned, the format and contents of reports can be modified, and new report types created. Additionally, user-defined templates to convert comma-delimited ASCII text files support importing of problems, incidents, and test logs.

Documentation and user support. The supplied documentation was well-written and complete. The Technical Reference Manual provides information to not only customize SQA:Manager, but to extend SQA:Manager functions and integrate them with other soft-

ware packages. SQA staff were both friendly and helpful. They answered all questions quickly.

Problems encountered. The installation of SQA:Manager was straightforward. No problems were encountered in its use.

20.3 Recent Changes and Planned Additions

A new version of SQA:Manager running under MS-Windows has been released. A version for the DEC Ultrix system is expected to enter beta testing soon.

20.4 Sample Outputs

Figures 20-1 through 20-18 provide sample outputs from SQA:Manager.

```
Test Plan Report - Current Version

Plan ID:    ACTIII02PN
Plan Name: AP & AR TEST PLAN
Description: TEST PLAN FOR VERSION 2.00 OF ENTIRE
             AP AND AR PROGRAMS IN ACT III PRODUCT.

Version:     2.0
Effective Date:    11/3/90
Author:            Beth Jones
Location:    C:OCSACT20.PLN

Referenced Specifications:

     ID: ACTIII02DS  Name: AP & AR DESIGN SPEC

Referenced Software Components:
        Product
        Name: ACT III  2.00

        Program
        Name: ACCOUNTS PAYABLE  2.00
        Name: ACCOUNTS RECEIVABLE    2.00

        Module
        Name: CHECK WRITER      2.00
        Name: INVOICE WRITER    1.96


Test Plan Report - Revision History

Plan ID:    ACTIII02PN
Plan Name: AP & AR TEST PLAN
Description: TEST PLAN FOR VERSION 2.00 OF ENTIRE
             AP AND AR PROGRAMS IN ACT III PRODUCT.
-------------------------------------------------------------------------------
Version:     1.5
Effective Date:    10/8/90

Author:            Beth Jones
Location:    DOCUMENTATION FILE CABINET
Version Description: FIRST VERSION
-------------------------------------------------------------------------------
Version:     2.0
Effective Date:    11/3/90

Author:            Beth Jones
Location:    C:OCSACT20.PLN
Version Description: CONTAINS NEW SECTIONS FOR TESTING
                     PRINTING OF CHECKS AND INVOICES
-------------------------------------------------------------------------------
```

Figure 20-1. SQA:Manager Test Plan for ACTIII02PN

```
Test Specification Report - Current

Test Specification ID: ACTIII02DS
Test Specification Name: AP & AR DESIGN SPEC
Description: TEST DESIGN SPEC FOR VERSION 2.00
             AP AND AR PROGRAMS IN ACT III

Version:          1.56
Effective Date:         12/6/90
Author:                 Beth Jones
Location:         C:OCSACT156.DSN
Version Description: SAME AS VER 1.50 PLUS FIXES FOR
                     COSMETIC PROBLEMS.

Referenced Procedures:
        ID: CHKRUNS     Name: CHECK GENERATION TESTS
        ID: INVRUNS     Name: INVOICE GENERATION

Referenced Test Cases:
        ID: CHKDATA     Name: CHECKING DATA PREP TESTS
        ID: CHKPOST     Name: CHECK POSTING TESTS
        ID: CHKPRN      Name: CHECK PRINTING TESTS
        ID: CHKRPT      Name: CHECK REPORT TESTS
        ID: CHKSEL      Name: CHECK SELECTION TESTS
        ID: CHKSRT      Name: CHECK SORTING TESTS
        ID: INV0001     Name: OPEN INVOICE REPORT TEST
                                  ...
        ID: INVPRN      Name: INVOICE PRINTING TESTS

Referenced Software Components:

        Program
        Name: ACCOUNTS PAYABLE 2.00
        Name: ACCOUNTS RECEIVABLE    2.00

        Module
        Name: CHECK WRITER      2.00
        Name: INVOICE WRITER    1.95



Test Specification Report - Revisions

Test Specification ID: ACTIII02DS
Test Specification Name: AP & AR DESIGN SPEC
Description: TEST DESIGN SPEC FOR VERSION 2.00
             AP AND AR PROGRAMS IN ACT III

-------------------------------------------------------------------------------
Version:          1.56
Effective Date:         12/6/90
Author:                 Beth Jones
Location:         C:OCSACT156.DSN
Version Description: SAME AS VER 1.50 PLUS FIXES FOR
                     COSMETIC PROBLEMS.
-------------------------------------------------------------------------------
```

Figure 20-2. SQA:Manager Test Specification Report for Test Spec ACTIII02DS

```
Test Case Report - Current

Test Case ID  INVPRN
Test case Name: INVOICE PRINTING TESTS
Description: TESTS FOR PRINTING INVOICES.

Tool ID:
Requirement ID:

Version:                1.2
Effective Date:         12/5/90

Developer:
Specification Location:          C:CASESPECSNVPRN.DOC
Version Description:  2ND VER - ADD NEW PAGE LENGTH ROUTINE
C:CASESNDATAINVPRN.IN
C:CASESRESULTSNVPRN.OUT

Referenced Software Components:
        Program
        Name: ACCOUNT PAYABLE     2.00

        Module
        Name: INVOICE WRITER   1.95


Test Case Report - Revisions

Test Case ID  INVPRN
Test case Name: INVOICE PRINTING TESTS
Description: TESTS FOR PRINTING INVOICES.

Tool ID:
Requirement ID:

-------------------------------------------------------------------------------

Version:                1.0
Effective Date:         9/10/90

Developer:              Mike Brown
Specification Location:          C:CASESPECSNVPRN.DOC
Version Description:  FIRST VERSION
C:CASESNDATAINVPRN.IN
C:CASESRESULTSNVPRN.OUT

-------------------------------------------------------------------------------

Version:                1.2
Effective Date:         12/5/90

Developer:
Specification Location:          C:CASESPECSNVPRN.DOC
Version Description:  2ND VER - ADD NEW PAGE LENGTH ROUTINE
C:CASESNDATAINVPRN.IN
C:CASESRESULTSNVPRN.OUT
```

Figure 20-3. SQA:Manager Test Case Report for Test Case INVPRN

```
Test Procedure Report - Current

Test Procedure ID  CHKRUNS
Test Procedure Name: CHECK GENERATION TESTS

Description: TESTS FROM ENTERING CHECK DATA TO
             POSTING  CHECKS.

Version:          2.0
Effective Date:         10/30/90

Developer:        Beth Jones
Procedure Location:
Version Description: MAJOR REV - ADDS PROCEDURES FOR CHECK
                     POSTING AND SELECTING TESTS.


Referenced Software Components:

        Program
        Name: ACCOUNTS PAYABLE 2.00

        Module
        Name: CHECK WRITER     2.00


Test Procedure Report - Revisions

Test Procedure ID  CHKRUNS
Test Procedure Name: CHECK GENERATION TESTS

Description: TESTS FROM ENTERING CHECK DATA TO
             POSTING  CHECKS.

-------------------------------------------------------------------------

Version:          1.0
Effective Date:         1/3/90

Developer:        Beth Jones
Procedure Location:
Version Description: FIRST VERSION

-------------------------------------------------------------------------

Version:          2.0
Effective Date:         10/3090

Developer:        Beth Jones
Procedure Location:
Version Description: MAJOR REV - ADDS PROCEDURE FOR CHECK
                     POSTING AND SELECTING TESTS.

-------------------------------------------------------------------------
```

Figure 20-4. SQA:Manager Test Procedure Report for Procedure CHKRUNS

```
Software Items Report
Date: 6/7/92

Product: ACT III  2.00
     Program: ACCOUNTS PAYABLE  2.00
          Module: DATABASE MANAGER  1.80
          Module: CHECK WRITER  2.00
          Module: JOURNAL  2.00
          Module: USER INTERFACE  1.80
     Program: ACCOUNTS RECEIVABLE  2.00
          Module: DATABASE MANAGER  1.80
          Module: INVOICE WRITER  1.95
          Module: JOURNAL  2.00
          Module: USER INTERFACE  1.80
     Program: GENERAL LEDGER  2.00
          Module: DATABASE MANAGER  1.80
          Module: JOURNAL  2.00
          Module: USER INTERFACE  1.80
```

Figure 20-5. SQA:Manager Software Items Report

```
Test Tool Report - List

Tool ID         Tool Name

AUTOT           AutoTester
ROBOT           SQA:Robot OS/2
ROBOT W         SQA:Robot Windows
SQAM            SQAManager
TESTPRO         TestPro


Test Tool Report

Tool ID:        TESTPRO
Tool Name:      TestPro
Version:        3.1
Vendor:         Sterling Software
Purpose:        Capture/Playback for MS-DOS

Date Acquired: 6/15/91
Location:
```

Figure 20-6. SQA:Manager Test Tool Report

```
Test Log Report

Test Log Name: ACT III VER 2.00 TESTING   Test Log ID: ACT3320

Ref'd. Test Procedure:
Description: Testing for Alpha Releases A-H
             of ACT III Ver 2.0

Test Cycle   Date      Time      Tester    Test Case ID   Incident ID
-------------------------------------------------------------------
A            12/26/90  08:00     MIKEB     CHKSRT         1
Garbage in upper left corner.
A            12/26/90  08:30     MIKEB     CHKPRN         2
Truncated Payee Names (over 40 chars)
A            12/26/90  09:00     MIKEB     CHKRPT         3
Memory problem.
A            12/26/90  10:00     MIKEB                    4
Error 202 when AP selected from Main Menu - intermittent
A            12/26/90  11:15     MIKEB                    5
Typo
A            12/26/90  15:00     MIKEB     CHKSRT         6
Checks don't appear in descending orderF
A            12/26/90  16:30     MIKEB     CHKSRT         0
Wrong system setup -tests aborted
A            12/27/90  08:00     MIKEB     CHKSRT         7
A            12/27/90  08:00     MIKEB     CHKSRT         0
Second run was OK - operator error (I think)
A            12/27/90  10:00     MIKEB     CHKDATA        10
A            12/27/90  11:00     MIKEB     CHKDATA        0
A            12/27/90  13:01     MIKEB                    9
404 Error
A            12/27/90  13:30     MIKEB     INV0001        0
A            12/27/90  14:00     MIKEB     INV0001        11
A            12/27/90  16:00     MIKEB                    12
A            12/27/90  16:05     MIKEB     CHKDATA        14
A            12/27/90  17:00     MIKEB     CHKDATA        0
A            12/27/90  17:15     MIKEB                    0
A            12/27/90  17:50     MIKEB                    17
A            12/27/90  18:00     MIKEB                    18
A            12/27/90  18:30     MIKEB                    20
A            12/28/90  08:00     MIKEB     INV0001        23
A            12/28/90  09:00     MIKEB     INV0001        0
A            12/28/90  19:00     MIKEB                    21
B            12/28/90  10:00     MIKEB     CHKSRT         0
B            12/29/90  08:30     MIKEB     CHKPRN         0
B            12/29/90  11:15     MIKEB     INVPRN         0
B            12/29/90  13:00     MIKEB     CHKDATA        0
```

Figure 20-7. SQA:Manager Test Log Report

```
Test Case Summary Report

Test Log Name: ACT III VER 2.00 TESTING   Test Log ID: ACT3320

Ref'd. Test Procedure:
Description: Testing for Alpha Releases A-H
             of ACT III Ver 2.0
Passed              : 9
Failed              : 9
Aborted             : 1
Incidents           : 17
Elapsed Time        : 21.30
```

Figure 20-8. SQA:Manager Test Case Report for Test Case INVPRN

Problems By Repair Person

Est Fix Date	Problem ID	Program Name	Asgn'd Dev.	Short Desc.
	3	ACCOUNTS PAYABLE: 2.00		Checksorting- wrong order
	5	ACCOUNTS PAYABLE: 2.00		Checkprinting- Can't use ind checks
	6	ACCOUNTS PAYABLE: 2.00		Control Account- selection error
	8	GENERAL LEDGER: 2.00		Help doesn't work
	10	ACCOUNTS RECEIVABLE: 2.00		
	11	ACCOUNTS RECEIVABLE: 2.00		Open Invoice Rpt- last one missing
	12	GENERAL LEDGER: 2.00		
	13	GENERAL LEDGER: 2.00		
	14	ACCOUNTS RECEIVABLE: 2.00		Postinvoices- missing posting date
	15	GENERAL LEDGER: 2.00	CATHYW	GL- won't accept MM/DD/YY format
1/8/91	2	ACCOUNTS PAYABLE: 2.00	CATHYW	Checkwriter- memory error.
1/8/91	7	ACCOUNTS PAYABLE: 2.00	CATHYW	Checkrecording- Tasks Menu - in loop
1/8/91	9	ACCOUNTS PAYABLE: 2.00	CATHYW	Chart of Accounts- Corrupted
1/15/91	1	ACCOUNTS PAYABLE: 2.00	CATHYW	Checkwriter- garbage
1/15/91	4	ACCOUNTS RECEIVABLE: 2.00	CATHYW	Checkdata- Unable to execute line 1232

Figure 20-9. SQA:Manager Problems Table

Fixed Problems Ready for Retest

Problem ID	Submitter's ID	Status	Fixed Program	Module/Subsystem	Short
1	MIKEB	Submitted			Check
2	MIKEB	Assigned	ACCOUNTS PAYABLE: 2.50	CHECK WRITER: 2.00	Check
3	MIKEB	Submitted			Check
4	MIKEB	Submitted			Check
5	MIKEB	Submitted			Check
6	MIKEB	Submitted			Contr
7	MIKEB	Assigned	ACCOUNTS PAYABLE: 2.50	USER INTERFACE: 1.80	Check
8	MIKEB	Submitted			Help
9	MIKEB	Assigned	ACCOUNTS PAYABLE: 2.50	DATABASE WRITER: 1.80	Chart
10	MIKEB	Submitted			
11	MIKEB	Submitted			Open
12	MIKEB	Submitted			
13	MIKEB	Submitted			
14	MIKEB	Submitted			Posti
15	MIKEB	Assigned	ACCOUNTS PAYABLE: 2.50		GL -

Figure 20-10. SQA:Manager Fixed Problems Ready for ReTest

Repair Cost

Program Name: ACCOUNTS PAYABLE
Version: 2.00

Cost of Repair

Date	Problem ID	Developer ID	Time (hours)	Cost
12/28/90	1	CATHYW		0.00
12/28/90	2	CATHYW	4.00	84.00
12/28/90	3			
12/28/90	5			
12/28/90	6			
12/28/90	7	CATHYW	1.00	21.00
12/28/90	9	CATHYW	2.00	42.00
Total				147.00

Figure 20-11. SQA:Manager Cost of Repair Table

Test Log Report

Test Log Name: ACT III VER 2.00 TESTING Test Log ID: ACT3320

Ref'd Test Procedure:
Description: Testing for Alpha Releases A-H
of ACT III Ver 2.0

Cost of Testing

Date	Time	Tester	Time (hours)	Cost
12/26/90	08:00	MIKEB	0.50	10.50
12/26/90	08:30	MIKEB	1.00	21.00
12/26/90	09:00	MIKEB	0.75	15.75
12/26/90	10:00	MIKEB	0.50	10.50
12/26/90	11:15	MIKEB	0.25	5.25
12/26/90	15:00	MIKEB	1.50	31.50
12/26/90	16:30	MIKEB	0.50	10.50
12/27/90	08:00	MIKEB	0.25	5.25
12/27/90	10:00	MIKEB	0.50	10.50
12/27/90	11:00	MIKEB	2.00	42.00
12/27/90	13:01	MIKEB	0.75	15.75
12/27/90	13:30	MIKEB	0.50	10.50
12/27/90	14:00	MIKEB	0.75	15.75
12/27/90	16:00	MIKEB	0.75	15.75
12/27/90	16:05	MIKEB	0.25	5.25
12/27/90	17:00	MIKEB	1.00	21.00
12/27/90	17:15	MIKEB	1.00	21.00
12/27/90	17:50	MIKEB	0.50	10.50
12/27/90	18:00	MIKEB	0.25	5.25
12/27/90	18:30	MIKEB	0.25	5.25
12/28/90	08:00	MIKEB	3.00	63.00
12/28/90	09:00	MIKEB	0.75	15.75
12/28/90	10:00	MIKEB	0.50	10.50
12/28/90	19:00	MIKEB	1.50	31.50
12/29/90	08:30	MIKEB	0.50	10.50
12/29/90	11:15	MIKEB	0.30	6.30
12/29/90	13:00	MIKEB	0.75	15.75
Total				442.05

Figure 20-12. SQA:Manager Cost of Testing

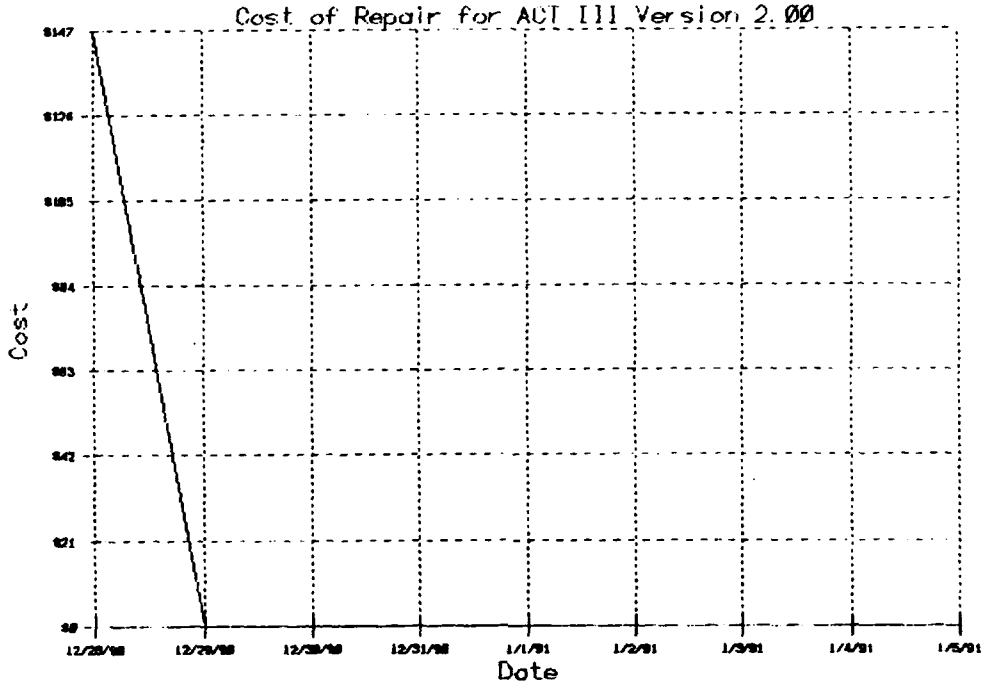


Figure 20-13. SQA:Manager Cost of Repair Graph

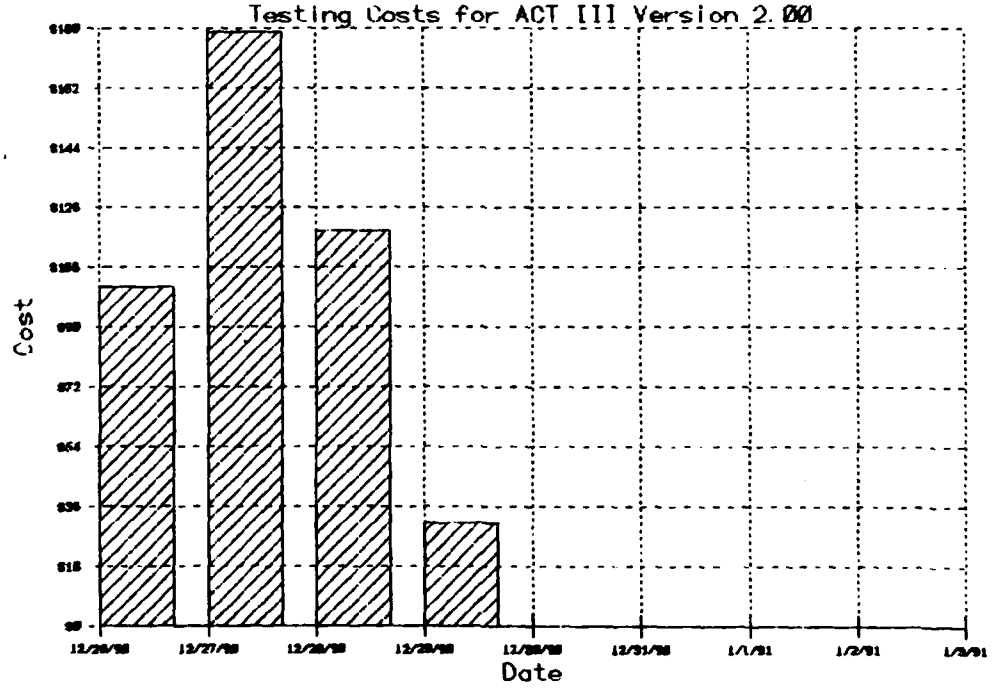


Figure 20-14. SQA:Manager Cost of Testing Histogram

Reliability Analysis
Poisson Geometric model

Date: 06/07/92

Program Name: TESTPROGRAM
Version: 1.00A

Total Testing Time (CPU hrs): 0
Total Failures Reported: 137
Target Reliability: 0.80 /1.0 CPU hours
Confidence Interval: 90%

Present Reliability
Low Limit: 0.00 /1.0 CPU hours
Most Likely: 0.00 /1.0 CPU hours
High Limit: 0.00 /1.0 CPU hours

Additional Time Needed for Execution

	CPU Time (hrs)	Calendar Time (days)
Low Limit:	224.8	936.9
Most Likely:	276.1	1150.4
High Limit:	357.5	1489.8

Additional Failures to be Found
Low Limit: 340 failures
Most Likely: 428 failures
High Limit: 568 failures

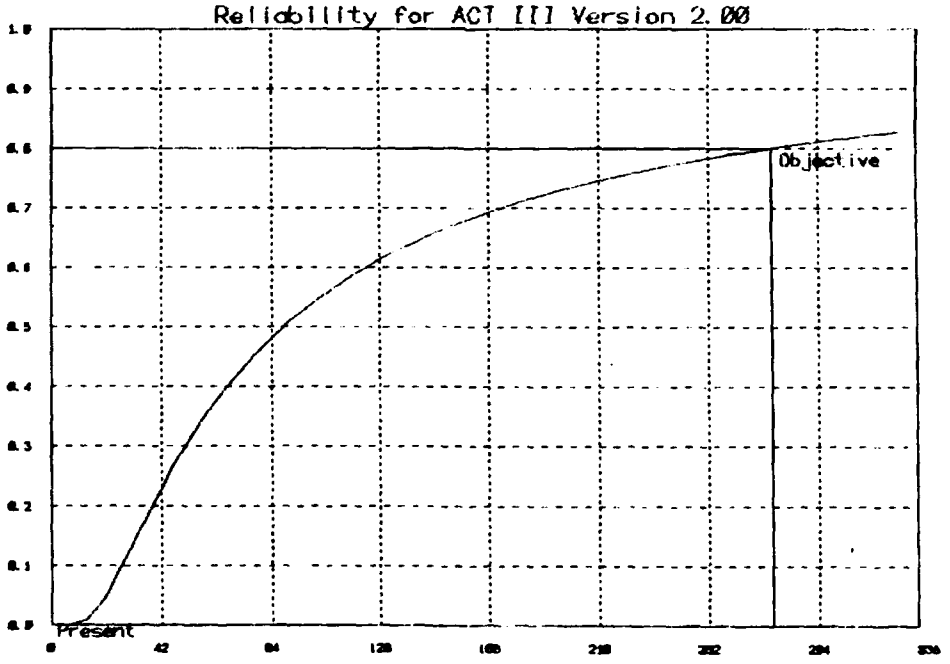


Figure 20-15. SQA:Manager Reliability Analysis Table and Graph

```
Failure Intensity Analysis
Poisson Geometric model

Date:        06/07/92

Program Name:      TESTPROGRAM
Version:     1.00A

Total Testing Time (CPU hrs): 0
Total Failures Reported:      137
Target Failure Intensity:     0.008000 failures/CPU sec
Confidence Interval:          90%

Initial Failure Intensity
        Low Limit:      0.422475 failures/CPU sec
        Most Likely:    0.588194 failures/CPU sec
        High Limit:     0.832898 failures/CPU sec

Present Failure Intensity
        Low Limit:      0.075974 failures/CPU sec
        Most Likely:    0.063783 failures/CPU sec
        High Limit:     0.054462 failures/CPU sec

Additional Time Needed for Execution
                CPU Time (hrs)    Calendar Time (days)
        Low Limit:      1.5             6.2
        Most Likely:    1.9             7.8
        High Limit:     2.5             10.3

Additional Failures to be Found
        Low Limit:      96 failures
        Most Likely:    128 failures
        High Limit:     180 failures
```

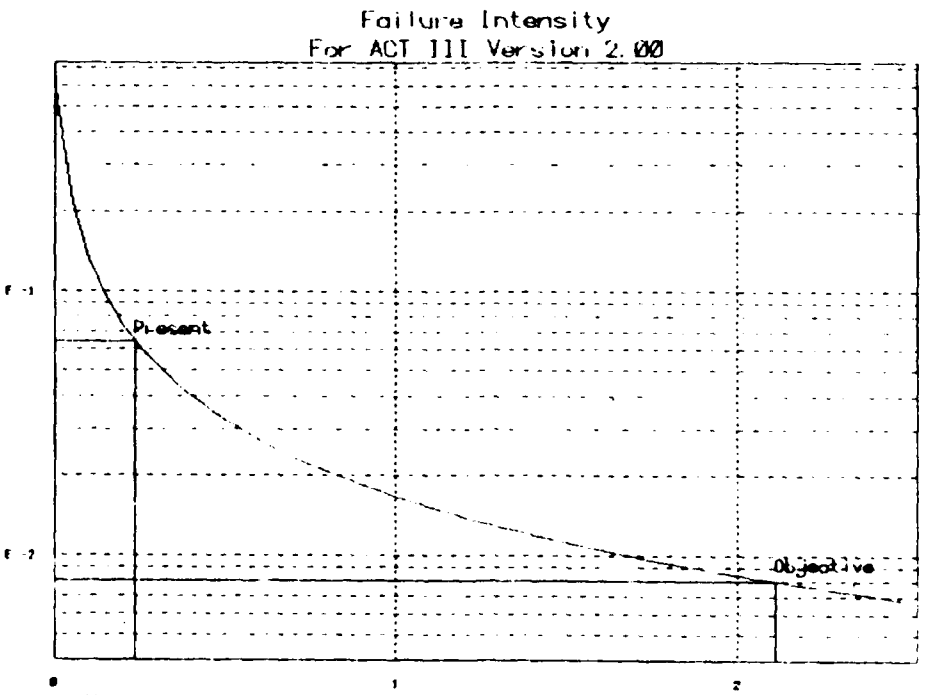


Figure 20-16. SQA:Manager Failure Intensity Table and Graph

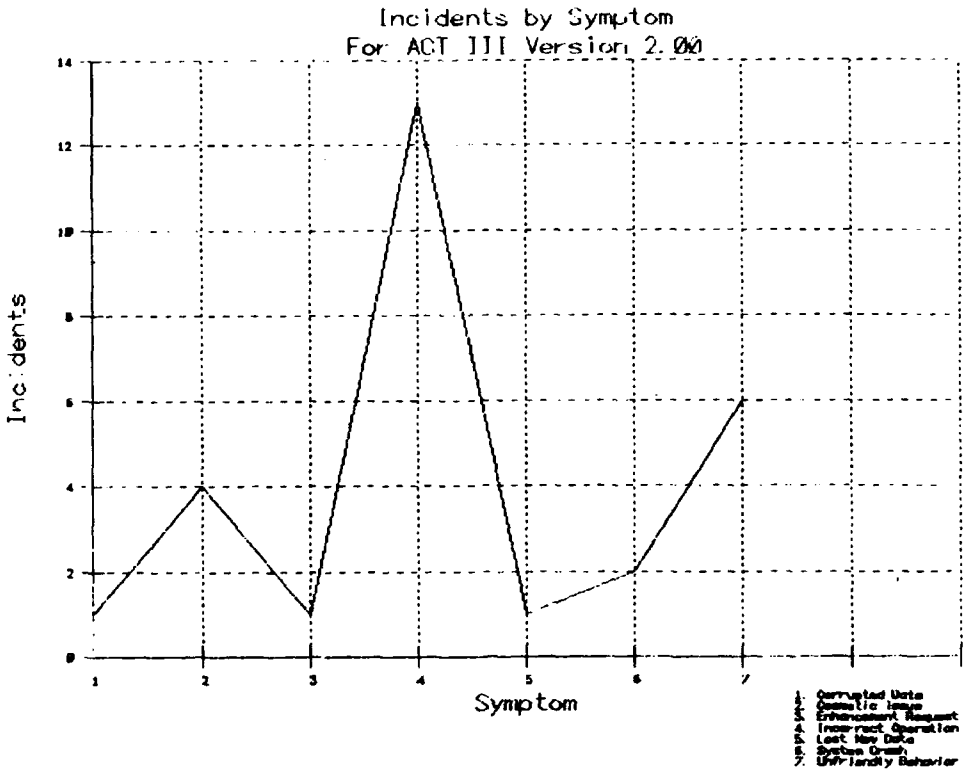


Figure 20-17. SQA:Manager Plot of Incidents by Symptom

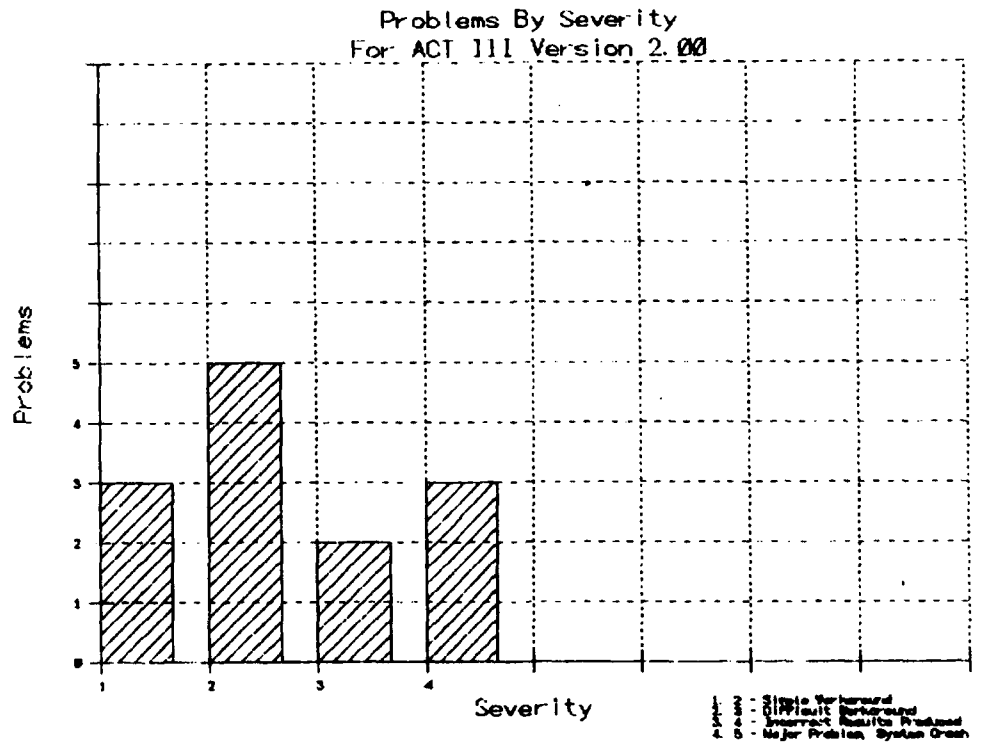


Figure 20-18. SQA:Manager Plot of Problems by Severity

21. SRE TOOLKIT

The SRE Toolkit supports reliability growth modeling using the Musa-Okumoto software reliability models. It takes a record of failure events, in terms of elapsed execution time from the start of test, and estimates various software reliability measures that track the progress of testing. This is particularly useful during system testing when the underlying faults causing failures are removed and hence "reliability growth" is occurring.

21.1 Tool Overview

The SRE Toolkit is available from Software Quality Engineering who hold a license from AT&T for this version of the AT&T software reliability engineering tools and accompanying training. The tools are provided as part of a consulting and training package. This package typically includes needs assessment and planning. Assistance in conducting and evaluating a pilot application is also available. The three day training courses are held at a public location or at a client's site. Attendance at a public course currently costs $995 per person, while up to 20 people can attend an on-site course for a total cost of $10,000.

The tool has been available since 1990. There are two versions. One runs under Unix System V on any supporting hardware. The other runs under DOS (release 3.3 onwards) on an AT&T PC or compatible. Both versions of the toolkit require the *awk* program. While screen graphics are available under MS-DOS, the Unix version of the tools generate *pic* commands for a graphics output device and, therefore, need the associated *pic* utilities. The evaluation was performed on the Unix version 3.11 of the toolkit.

The two main tools in the toolkit are *est*, the reliability estimation tool, and *plot*, the graphics support tool. For *est*, the user specifies whether an exponential or logarithmic reliability model is required and the failure intensity objective that will be used to determine when to stop testing. He can specify whether failure data entries should be interpreted to correspond to individual failure events, or to the number of failures that occurred since the previous failure entry or start of test interval (grouped data analysis). A testing compression factor specifies the desired ratio of field execution time to test execution time allowing, for example, more stress to be placed on in-house testing than field testing. In the case of the exponential model, the user can also specify a failure time adjustment to adjust failure times to take into account the incremental delivery of software to system test. For the logarithmic model, a failure intensity decay parameter determines the rate of exponential decay.

The *est* tool uses the fitted model to estimate several reliability measures over a range of confidence limits. These reliability measures include the present failure intensity and the additional failures, test execution time, and work days required to meet the specified failure intensity objective. In the case of an exponential reliability model, the total number of failures that would be experienced after an infinite amount of execution time is also given. Finally, the expected calendar date when the failure intensity objective will be met and the additional calendar time needed for testing that this implies are reported. In addition to such tabular data, *est* generates a file of plot commands for graphic output.

Prototype tools are included in the toolkit:

- *resrusg*. Uses simple regression analysis to estimate the testing resource usage parameters. It produces summary statistics for each recording period and estimates showing the resources consumed per unit execution time and resources consumed per failure parameter. *resrusg* also generates plots that show how well the regressions fit the original data.
- *reldem*. Plots reliability demonstration charts that indicate, for a given failure intensity objective, whether testing should continue.
- *predat*. Plots the completion date for testing, that is, when the failure intensity objective is met versus failure intensity objective. It also produces a table indicating testing periods where a particular testing resource is a limiting resource.
- *minfio*. Plots the total life cycle costs versus failure intensity objective. It includes the system test and operational life cost.
- *logmod*. Plots the logarithmic reliability growth model using model parameters, initial failure intensity, and the failure intensity decay factor.
- *expmod*. Plots the exponential reliability growth model using model parameters, initial failure intensity, and total failures after infinite execution time.

21.2 Observations

Ease of use. The tools are written as shell scripts (or batch files in the case of the MS-DOS version). Their operation can be tailored using parameters that are given in a parameter file or, in some cases, included with a tool's input data. The parameters range from specifying the formatting of an output plot to such details as the cost per failure identification resource hour. They provide considerable flexibility in the application of each tool.

Documentation. The documentation provided in the form of Unix-like *man* pages is both helpful and extensive. The definition of the file format for failure data facilitates importing data. Installation was straightforward although minor problems arose due to system dependencies.

Problems encountered. The tools operated as described in the documentation.

21.3 Sample Outputs

Figures 21-1 through 21-9 provide sample outputs from SRE Toolkit.

```
FAILURE PARAMETER FILE IS tst_stg.fp
FAILURE TIME ANALYSIS WILL BE DONE
TEST COMPRESSION FACTOR OF 15.1 WILL BE APPLIED DURING ANALYSIS
FAILURE TIME FILE IS tst_stg.ft
FAILURE TIMES WERE ADJUSTED
ADJUSTED FAILURE TIMES FILE IS tst_stg.ad
GENERATING OUTPUT REPORT

                    SOFTWARE RELIABILITY ESTIMATION
                       EXPONENTIAL (BASIC) MODEL
                TST DATA SET WITH STAGED DEVELOPMENT ADJUSTMENTS

BASED ON SAMPLE OF              136 TEST FAILURES
TEST EXECUTION TIME IS          21.4559 CPU-HR
FAILURE INTENSITY OBJECTIVE IS  2.4 FAILURES/1000-CPU-HR
CURRENT DATE IN TEST            861109
TIME FROM START OF TEST IS      96 DAYS

                      CONF. LIMITS          MOST          CONF. LIMITS
                 95%    90%    75%    50%  LIKELY   50%    75%    90%    95%
                ------ ------ ------ ------ ------ ------ ------ ------ ------
TOTAL FAILURES     138    139    139    140    141    142    144    145    147
     ********** FAILURE INTENSITIES (FAILURES/1000-CPU-HR) **********
INITIAL           1190   1234   1303   1370   1467   1566   1636   1710   1757
PRESENT          28.59  31.35  36.19  41.49  50.20  60.54  68.95  78.78  85.67
     *** ADDITIONAL REQUIREMENTS TO MEET FAILURE INTENSITY OBJECTIVE ***
FAILURES             2      2      3      3      5      6      7      9     10
TEST EXEC. TIME  12.91  13.79  15.28  16.84  19.33  22.20  24.52  27.23  29.15
WORK DAYS         3.05   3.29   3.70   4.15   4.95   5.97   6.87   8.00   8.85
COMPLETION DATE 861113 861113 861113 861114 861114 861117 861118 861119 861120

GENERATING PLOT COMMANDS. COMPLETED! PLOT COMMAND FILE IS tst_stg.pc.
```

Figure 21-1. SRE Toolkit Generated Reliability Measures

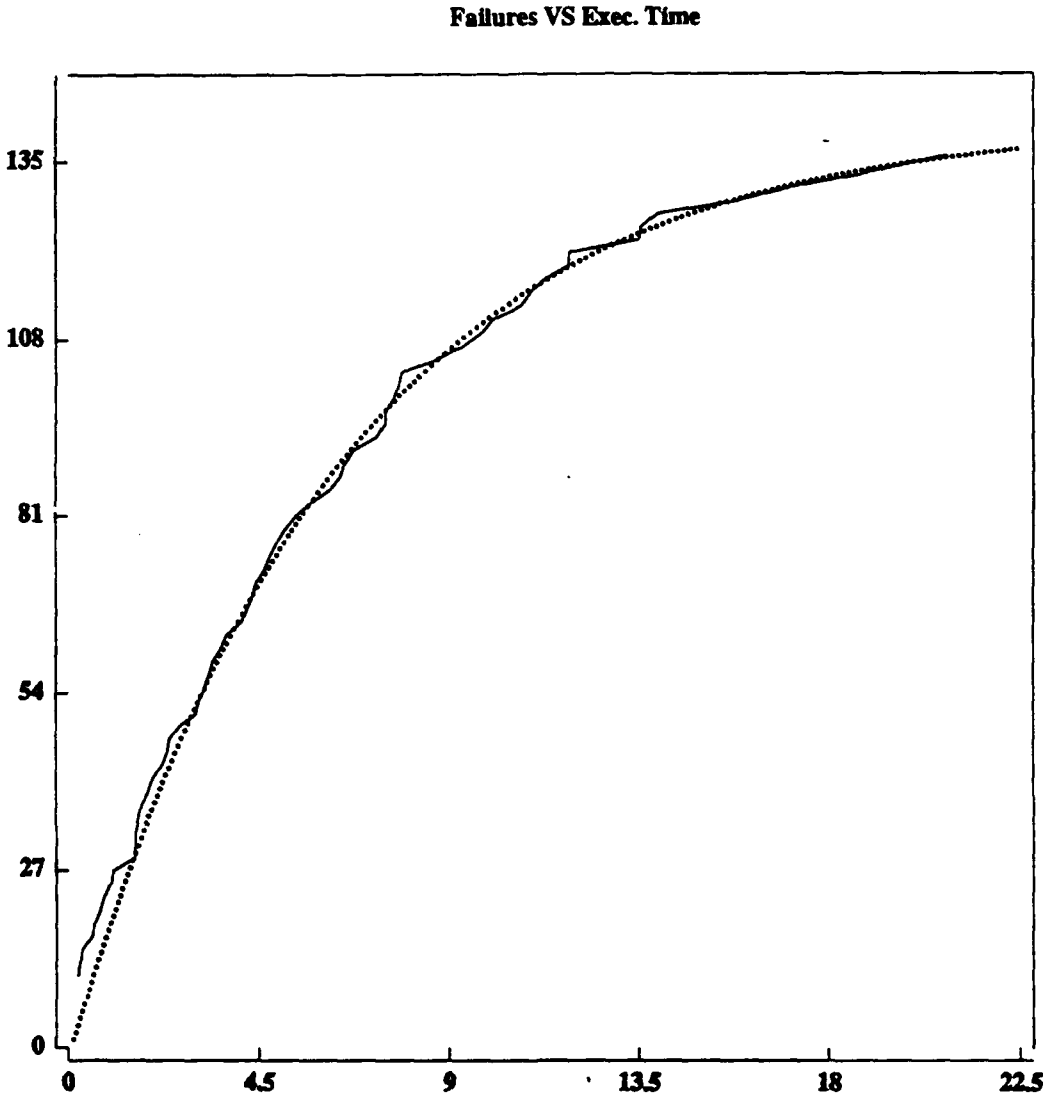


Figure 21-2. SRE Toolkit Failure vs. Execution Time Plot

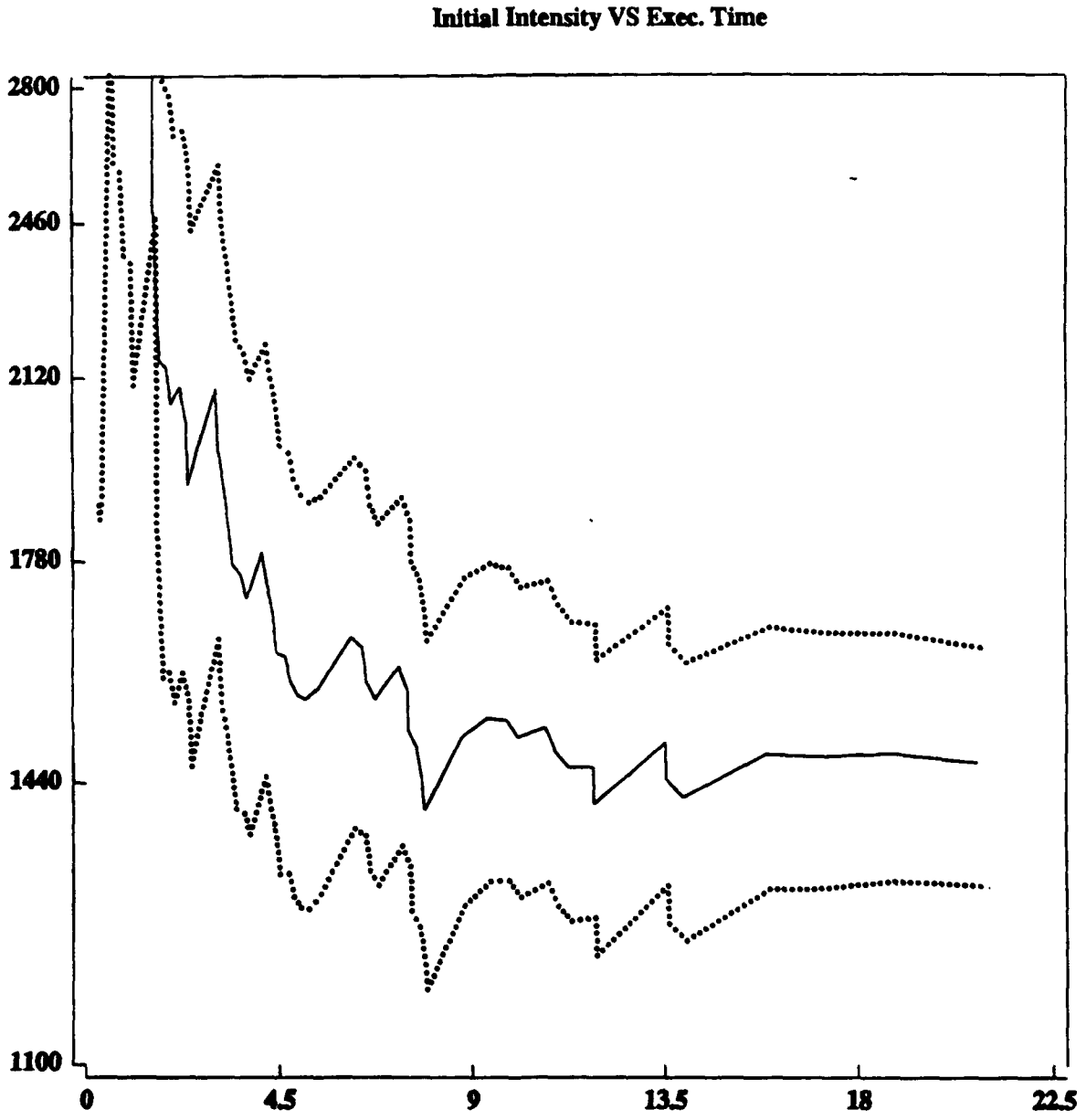


Figure 21-3. SRE Toolkit Initial Intensity vs. Execution Time Plot

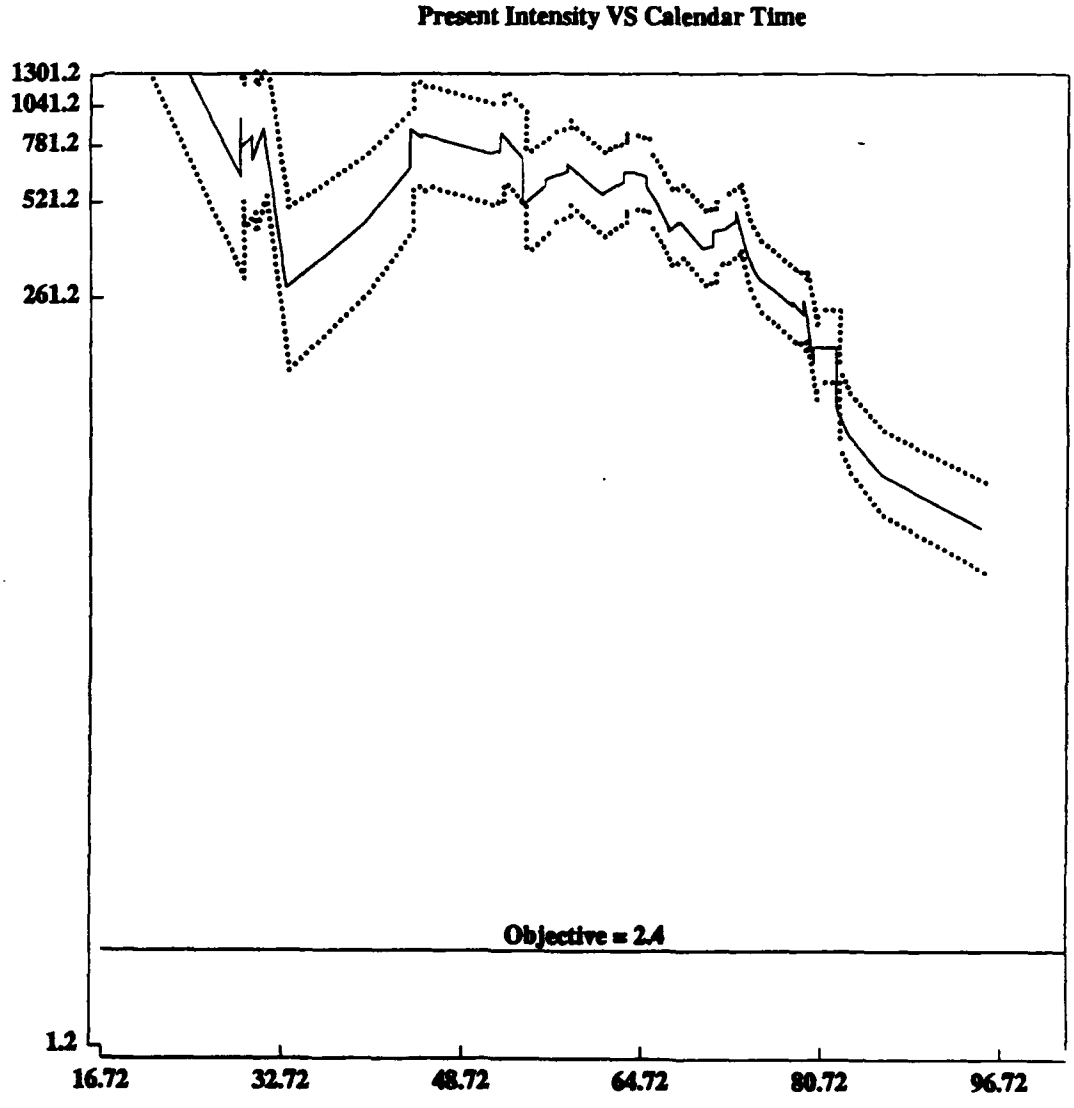


Figure 21-4. SRE Toolkit Present Intensity vs. Calendar Time Plot

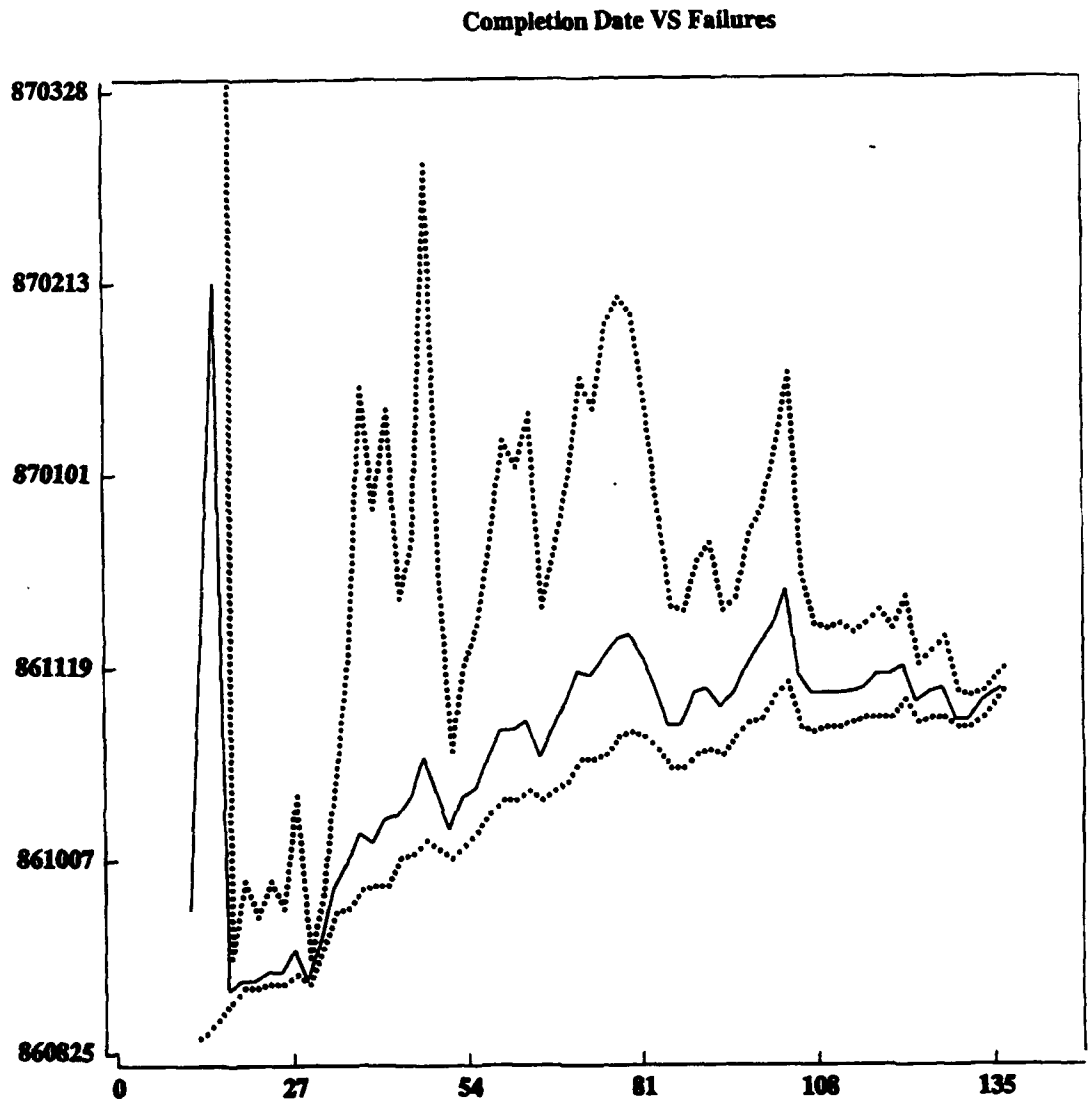


Figure 21-5. SRE Toolkit Completion Date vs. Failure Data

Period	Ident. Rate	Computer Rate	Appr. Fail.Int.	Corr. Work	Items Corr.
1	26.8949	76.5281	39.1198	50.6	7
2	15.4162	25.7965	13.3607	87.5	6
3	22.3404	29.7872	27.6596	59	5
4	16.3866	31.9328	10.9244	170	19
5	24.2537	16.4179	10.4478	103.1	19
6	12.6506	11.8976	5.42169	66	19
7	6.17978	7.30337	3.37079	92.7	10
8	6.39098	9.58647	4.51128	105.5	12
9	4.90716	8.32891	3.97878	75.1	20
10	5.13158	6.28947	1.31579	105.1	19

```
Failure Identification Resources
thetai = 7.19208 hrs/CPU-hr, mui = 0.571399 hrs/failure
Failure Correction Resources
muf = 6.05505 hrs/fix
Computing Resources
thetac = 2.93493 hrs/CPU-hr, muc = 1.6195 hrs/failure

Reformatted input is in file ex6b.out
```

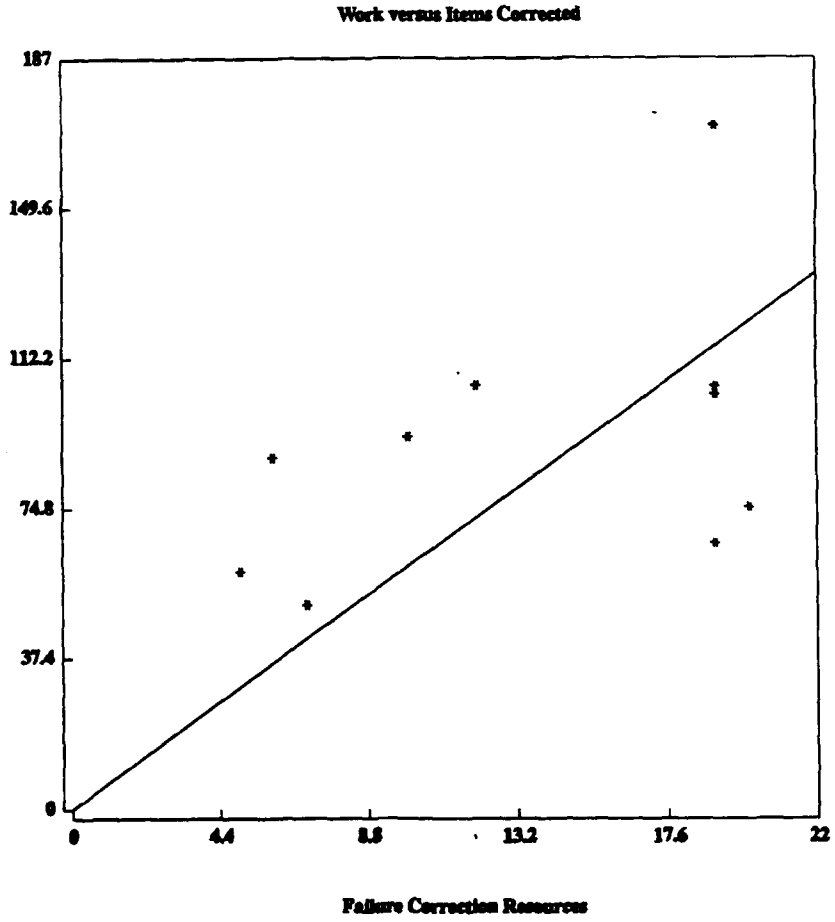


Figure 21-6. SRE Toolkit Testing Resource Usage Parameter Estimation

LIMITING RESOURCE PERIODS

DATE	EX.TIME CPU-HR	FAIL.INT. F/KCPU-HR	LIMITIING RESOURCE
901001	0	1000	
			FAILURE CORRECTION
901224	752.039	222.222	
			FAILURE IDENTIFICATION
999999	999999	0	

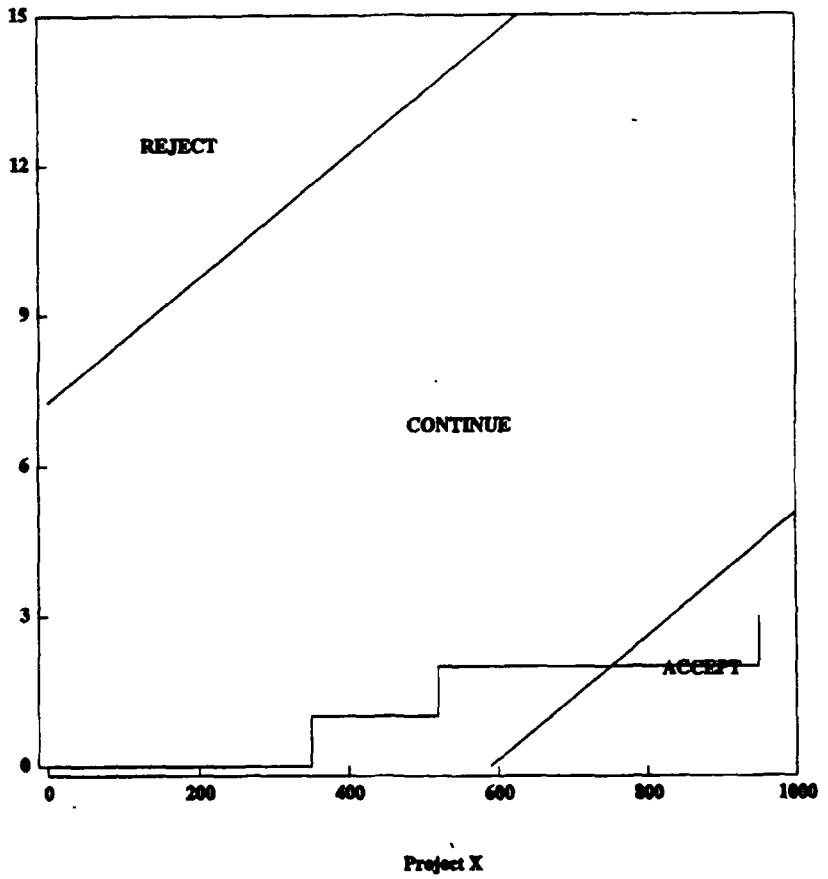


Figure 21-7. SRE Toolkit Reliability Demonstration Chart

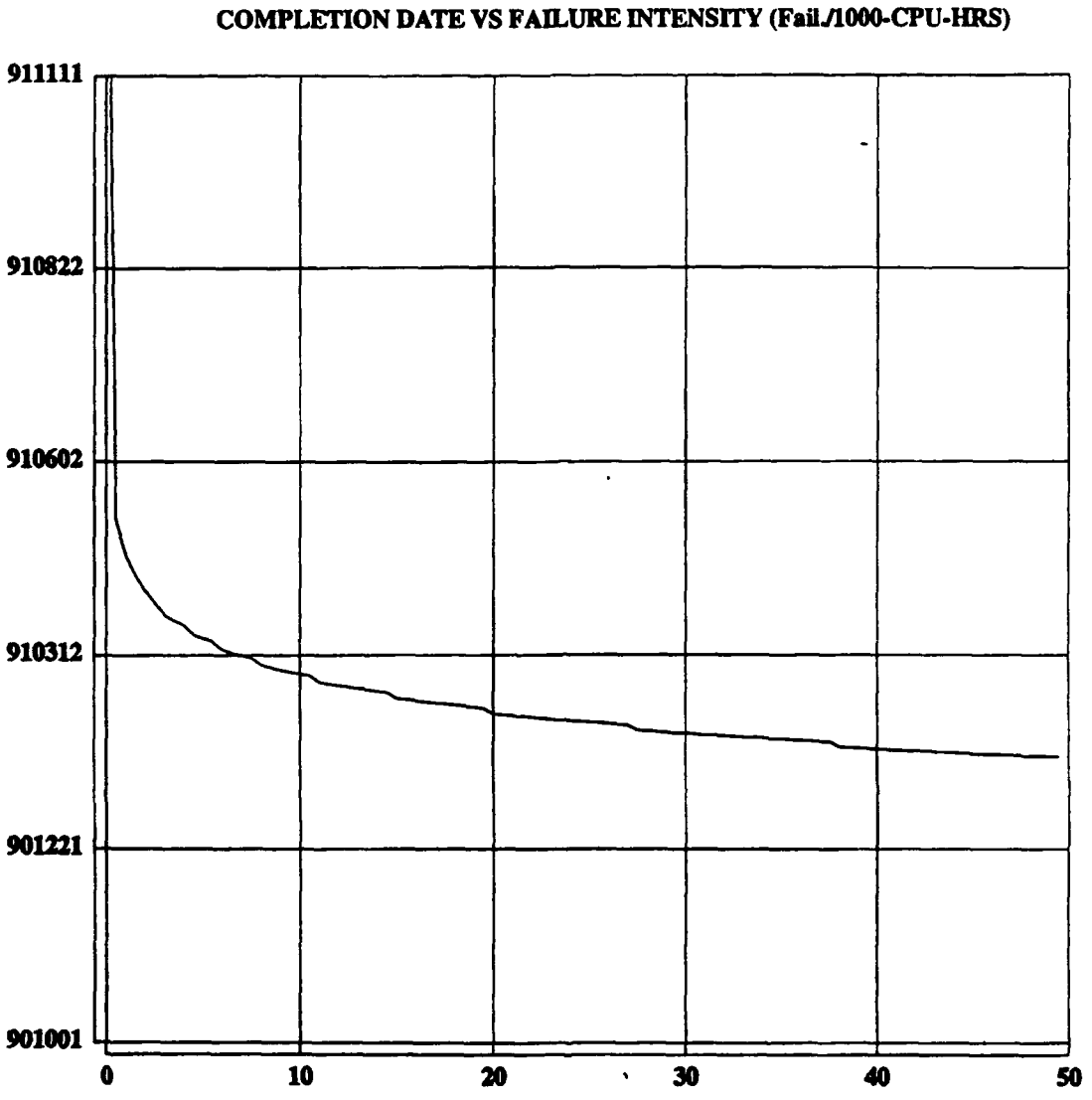


Figure 21-8. SRE Toolkit Completion Date vs. Failure Intensity Output

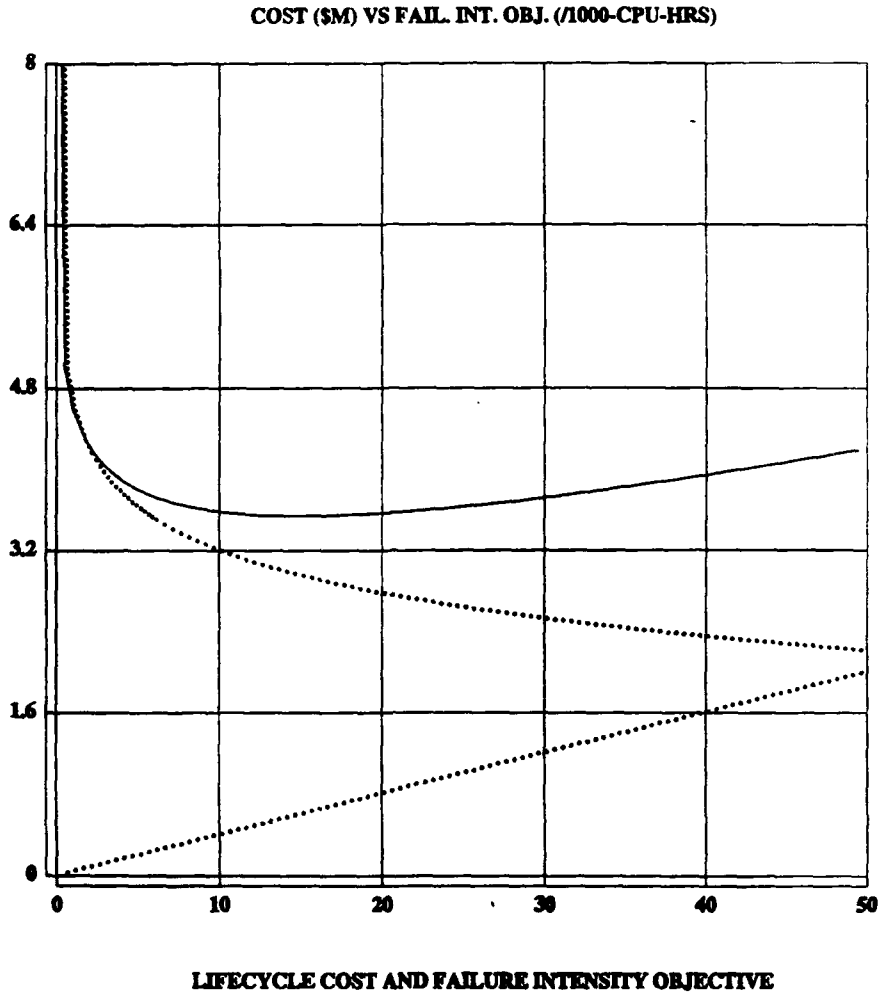


Figure 21-9. SRE Toolkit Life Cycle Cost and Failure Intensity Objective Plot

22. T

T generates test data from requirements information and automatically provides tracing between tests and defined software actions. Its goal is to generate the minimum number of traceable, unique test cases that will exercise every operation and each of a set of vendor-defined probable errors at least once. Test adequacy is assessed based on requirements coverage, input domain coverage, output range coverage and, optionally, structural coverage. T can be used during any software development stage; during maintenance test data is generated for software changes only. T is already used by various government organizations, including the Naval Avionics Center, the Jet Propulsion Laboratories, Naval Coastal Systems Center, and U.S. Army Forts Monmouth and Sill.

Runner is a companion tool that provides test capture/playback for C.

22.1 Tool Overview

T was developed by Programming Environment Inc. In addition to T and Runner, this organization markets consultancy and training services, and supports tool users with a quarterly newsletter. T has been available since 1987 and has over 1,000 users. It is available for PC/MS-DOS and VAX/VMS platforms, and various workstations under Unix. Training is a prerequisite for tool purchase and costs $1,500 per person at Programming Environments, Inc. or $10,000 for an on-site workshop. At the time of evaluation, prices for T itself started at $7,000. T, and training materials, are, however, provided free to any university that teaches courses on software testing.

Interfaces between T and some leading CASE tools are available. An interface to Teamwork is marketed by Cadre Technologies, Inc. and Interactive Development Environments markets an interface to StP. (Boeing has developed a proprietary interface for the Excelerator CASE and Texas Instruments for their Information Engineering Facility.) Support for reverse engineering of TSDL specifications from existing code is also available. In this case, Cadre markets tools to reverse generate a substantial part of TSDL specifications from Ada, C, or Fortran. (Boeing is developing a proprietary tool providing the same function for COBOL. Boeing is also developing tools to generate Ada and C code from TSDL specifications.) Code generation may also be supported by IBM's current effort to convert from TSDL to Z.

To support test execution, interfaces between T and the AutoTester Corp. product AutoTester, the Mercury Interactive Corp. product XRunner, and Tiburon's Ferret capture/playback tools are commercially available.

The examination was performed on T version B3.0, running on a Sun SPARC workstation under Unix. This version was a demonstration copy of T, fully functional with the exception that only a limited number of data definitions and sentences can be processed at one time and some reports are not generated. Additionally, the full set of test design rules is proprietary information only available with purchase of full version; as a consequence, the sampling rules cannot be changed and features such as random sampling are not available in the demonstration version.

Before using T, the user must prepare a description of software actions, data, events, and condition states. At the system level, this type of requirements information may be derived from sources such as data flow diagrams, state transition diagrams, entity relationship diagrams, and control diagrams. At the design or unit level, module descriptions will be the primary source. The description is specified in the T Software Description Language (TSDL), a superset of the Semantic Transfer Language (STL) [IEEE 1992]. TSDL allows specification of operation statements and definitions for data items, conditions, and states. It also supports event types with multiple inheritance for operations supported by temporal conditions. Standard templates are provided to help write TSDL descriptions. The collection of resulting ASCII files is called a *Software Description File (SDF).*

Once an SDF has been prepared, the first component of T, called Tverify, can be run. This translates the SDF into a *Software Description Data Base (SDDB)* and then checks this database for syntax errors and to see whether it contains the necessary and sufficient information for test case design and preparation. This verification also provides metrics such as counts of actions, states, conditions, events, and data items defined in the SDF, and error counts. A cross-reference report shows where every item is used (this report is not available with the demonstration version).

After successful verification, the user can request T to design test cases. Here the second T component, called Tdesign, takes information from the SDF to group actions into appropriate states. Test design rules are applied to partition data item domains such that all of the values of the data item in a partition will probably be processed in the same manner. The test design rules used by T are derived from the following test techniques:

- Functional testing. The input domain is partitioned into classes such that each member of a class causes a given action to be executed. Test data is selected from these partitions that will cause all actions to be exercised.
- Equivalence class partitioning. The input domain is partitioned into a number of equivalence classes such that a test of a representative value of each class is equivalent to a test of any other value. The minimum set of test data that will invoke as many different input conditions as possible is selected from these partitions.
- Boundary value analysis. The input domain is partitioned into a number of equivalence classes. Test data is chosen that lies at the edge of these partitions to reflect the boundaries of the input domain.
- Cause-effect graphing. The input and output domains are partitioned into classes that specify which input classes cause which outputs to occur. Test data is selected that will cause all effects to be exercised.
- State-directed testing. Test data is selected to cause every transition between states to be exercised.
- Event-directed testing. Test data is selected to cause every event (some signal passing from the external world) to be exercised.

A fault-directed approach is used to select samples from the partitions yielded by the above techniques. This approach guides the selection of both valid and invalid samples that are likely to detect errors. By default T selects the following samples:

- *For the valid subdomain:*
  a. Low boundary value,
  b. Just above low boundary value,
  c. Reference value (midway between the low boundary and high boundary values),
  d. Out-of-type values 1 to n,
  e. Just under high boundary value, and
  f. High boundary value.
- For the invalid subdomain:
  a. Just below low boundary value, and
  b. Just above high boundary value.

The actual values taken, of course, depend on the type of the data item in question. Out-of-type samples for an integer, for example, are a decimal, single character, and list of character values. As appropriate, additional samples are taken; for integers that include the value zero, for example, a troublesome sample is taken, with value zero. Though not experienced in the course of this examination, the user can define his own samples and modify the sampling rules. For example, the user can add additional normal values and abnormal values.

A technique called *guided synthesis* is used to combine samples. This method provides a repeatable strategy for probing a two dimensional testing space where only one data item is varied at a time, the others being held at a normal, or reference, value.

Tdesign also creates the *Test Case Data Base (TCDB)* which contains one file per test, designed for easy use by test execution tools. Test design metrics provide all the non-subjective values used for function point calculation. (The test design metrics report is not available with the demonstration version. The trace report that provides a cross-reference between software actions and tests is similarly not available; this information is, however, included on a case-by-case basis in the test case summary report that details each generated test case.)

T supports test execution by providing a model for calculating test coverage. T treats test coverage as including requirements, input domain, output range, and structure coverage. These factors are weighted individually and summed to produce an overall *Testing Comprehensiveness* (TC) measure. The user can adjust the weights to, for example, reflect different priorities at unit, integration, and system test levels. Requirements and input domain coverage are automatically reported by T. Currently, the output range and structure coverage must be recorded manually, and TC calculated manually. An on-line pass/fail recording facility is under development that will allow a user to specify test priorities and record dates of execution and test results to allow automating this calculation.

22.2 Observations

Ease of use. The demonstration version of T is easy to install and use; the difficult part lies in creating the TSDL description. Although the demonstration version is limited to a character-based menu user interface, the full version of T allows users a choice between this and a graphical user interface, command line interface, or script interface. Context-sensitive, multi-level on-line help is available.

Documentation and user support. The documentation provided with the demonstration version of T included a full description of TSDL and was adequate for its use. Programming Environments, Inc. were very helpful in answering questions.

Problems encountered. The demonstration version operated as described in the documentation.

22.3 Recent Changes and Planned Additions

A new component is now being delivered with T. Called Tprepare, this component provides for rule-based preparation of test documentation. The user defines his own rules allowing, for example, documentation to conform to DoD or IEEE standards.

Several new features are expected to be released in the first quarter of 1993. Torder, a new T component, will order test cases and write test scripts to handle the necessary test set up and clean up. Another component, Quantifier, will take user-supplied test results to automatically calculate and report on the TC coverage measure. T will also support generation of an operational profile to support Musa's reliability assessment.

An interface to the AutoTester test execution tool is under development.

22.4 Sample Outputs

Figures 22-1 through 22-7 provide sample outputs from T.

```
                                ...
/* This Software Description File, SDF, contains a partial        */
/* specification. There is enough information in this SDF to       */
/* illustrate the STL and the basic processing in T. There is      */
/* not enough information in this SDF to specify a complete        */
/* lexical analyzer generator or to demonstrate T completely.      */
                                ...
S_packet                                    Adalex_1
   has subject                              "Adalex_specification" ;
   has content version                      "1";
   has description                          "This S_Packet sentence          "
                                            "identifies the set of           "
                                            "information in this file.       "
   .

#include <tsdl.std>  /* This line includes standard definitions          */
                     /* The standard definitions will help this sdf,     */
                     /* but they will also cause some extra definitions  */
                     /* to appear in the report called verify.rpt        */

/* ------------------------------------------------------------------------*/
/* ---------- Begin Action Definitions ----------------------------------*/
/* ------------------------------------------------------------------------*/

/* ------ The words, identifier and integer, are keywords in IEEE 1175 ----*/
/* ------ so they definition will cause T to generate comments. However ---*/
/* ------ there are no reserved words in the T scanner/parser so the ------*/
/* ------ key words will be processed correctly. --------------------------*/

Action                                      Adalex
   is actiontype                            internal;
   is selected by                           "invocation"  ;
   is concluded on                          "termination" ;
   uses dataitem                            context_and_lexicon ,
                                            identifier,        /* Patterns */
                                            letter,            /* Patterns */
                                            digit,             /* Patterns */
                                            letter_or_digit,   /* Patterns */
                                            integer,           /* Patterns */
                                            decimal_literal,   /* Patterns */
                                            operator_symbol,   /* Patterns */
                                            left_parenthesis,  /* Patterns */
                                            right_parenthesis, /* Patterns */
                                            combining_option ;
   produces dataitem                        a_new_lexical_analyzer,
                                            Standard_Error_File ;
   has description                          "The scanner produced depends upon "
                                            "three items: 1- A data type, a    "
                                            "table, defining tokens, 2- A      "
                                            "get_next_character procedure, and"
                                            "3- A make_token procedure.        " ;
   .
```

Figure 22-1. T Sample SDF

```
/* ------------------------------------------------------------------------*/
/* ------------ Begin Data Definitions ------------------------------------*/
/* ------------------------------------------------------------------------*/

Dataitem                                    context_and_lexicon
   has placeholder value                    "This ishould be the input context an
   has description                          "On page 5 of IDA Paper P-2100     "
                                            "the specification for Adalex      "
                                            "names a context_clause, a         "
                                            "generic_formal_part, a separate   "
                                            "parent_name and a lexicon. The    "
                                            "named items are not defined in    "
                                            "P-2100 so they are illustrated    "
                                            " in this file with a placeholder  "
                                            "or to-be-defined value." .

Dataitem identifier is an instance of datatype identifier_t .
Datatype                                    identifier_t
   is datatypeclass                         string;
   has values                               //[A-Za-z]1,64_[A-Za-z0-9]1,64// ;
   has valid subdomain                      as_specified;
   has invalid subdomain                    abnormal;
   has description                          "P-2109 did not define the        "
                                            "allowed minimum or maximum       "
                                            "of identifiers. So a minimum of  "
                                            "one character and a maximum of   "
                                            "128 characters was assumed.      "
   .

Dataitem letter is an instance of datatype letter_t .
Datatype                                    letter_t
   is datatypeclass                         string;
   has values                               //[A-Za-z]// ;
   has valid subdomain                      as_specified;
   has invalid subdomain                    abnormal;
   .

Dataitem digit is an instance of datatype digit_t .
DataType                                    digit_t
   is datatypeclass                         character ;
   has values                               //[0-9]// ;
   has valid subdomain                      as_specified;
   has invalid subdomain                    abnormal.

Dataitem letter_or_digit is an instance of datatype letter_or_digit_t .
DataType                                    letter_or_digit_t
   is datatypeclass                         character;
   has values                               //[A-Za-z0-9]//;
   has valid subdomain                      as_specified;
   has invalid subdomain                    abnormal.
```

Figure 22-1 continued: T Sample SDF

```
Dataitem integer is an instance of datatype integer_t .
DataType                                integer_t
   is datatypeclass                     character;
   has values                           //[0-9]1,64//;
   has valid subdomain                  as_specified;
   has invalid subdomain                abnormal.

Dataitem decimal_literal is an instance of datatype decimal_literal_t .
DataType                                decimal_literal_t
   is datatypeclass                     string;
   has values                           //[0-9]1,16.[0-9]0,16//;
   has valid subdomain                  as_specified;
   has invalid subdomain                abnormal.

Dataitem operator_symbol is an instance of datatype operator_symbol_t .
Datatype                                operator_symbol_t
   is datatypeclass                     string;
   has values                           "+", "-", "*", "/";
   has valid subdomain                  as_specified;
   has invalid subdomain                abnormal;
   .
Dataitem                                left_parenthesis
   has fixed value                      "(";
   .
Dataitem                                right_parenthesis
   has fixed value                      ")";
   .

Dataitem combining_option is an instance of datatype combining_option_t.

Datatype                                combining_option_t
   is datatypeclass                     string;
   has values                           "is copied", "is separate", "is generic" ;
   has valid subdomain                  as_specified ;
   has invalid subdomain                abnormal ;
   has description                      "The combining option tells Adalex"
                                        "how to package the generated     "
                                        "scanner."
   .

Dataitem                                a_new_lexical_analyzer
   has fixed value                      "Ada source code for a scanner" ;
   .
Dataitem                                Standard_Error_File
   has fixed value                      "A report of errors detected"
   .

Dataitem                                EOS
   has fixed value                      "EOS is TRUE when the last character is reached
   has description                      "EOS means End_of_stream. EOS is "
                                        "FALSE as long as there are more "
                                        "characters in the input stream. ";
                                ...
```

Figure 22-1 continued: T Sample SDF

```
T Software Description Verification Version 3.0 (Restricted)
Copyright (C) 1987-1992 Programming Environments, Inc.
================================================================================
                                Translation
================================================================================
    1  #line  1  "sdf"
                                    ...
   12  /* This Software Description File, SDF, contains a partial        */
   13  /* specification. There is enough information in this SDF to      */
   14  /* illustrate the STL and the basic processing in T. There is     */
   15  /* not enough information in this SDF to specify a complete       */
   16  /* lexical analyzer generator or to demonstrate T completely.     */
                                    ...
   21  S_packet                                  Adalex_1
   22     has subject                            "Adalex_specification" ;
   23     has content version                    "1";
   24     has description                        "This S_Packet sentence          "
   25                                            "identifies the set of           "
   26                                            "information in this file.       "
   27     .
   28
   29  #line  1  "/eval/tcode/tsdl.std"
    1  #noecho
  142  #line  30  "sdf"
   30                        /* The standard definitions will help this sdf,    */
                                    ...
  174  Dataitem                                  next_character
  175     has fixed value                        "The output character should equal the i
  176     .
  177
      <End of File>

 - finished translation with  46  recognisable TSDL sentences out of  46
 - saving description data base

translator messages
   4 comment messages

================================================================================
                                Interpretation
================================================================================
s_packet: Adalex_1
unitdate: Fri Oct  9 13:13:49 1992

================================================================================
                                Evaluation
================================================================================
Accept  Reject  Name
------  ------  ----------------------------------------
 ***             Adalex
```

Figure 22-2. T Software Description Verification

```
Extra dataitems
   EOS
   next_character

Extra datatypes
   Bit
   Bit8string
   Digit
   LocalPhone
   PrintableASCII
   USPhone
   ZipCode5
   ZipCode9
   boolean_integer
   boolean_string

Extra states
   <none>

Extra statetransitions
   <none>
```

Figure 22-2 continued: T Software Description Verification

```
T Software Description Metrics  Version 3.0
Copyright (C) 1987-1992 Programming Environments, Inc.

s_packet: Adalex_1
unitdate: Fri Oct  9 13:13:49 1992
```

Total	Extra	Verified	Unverified	Dynamic
1	0	1	0	action(s)
0	0	0	0	statetransition(s)

Total	Extra	Verified		Unverified		Static
		In	Out	In	Out	
0	0	0	0	0	0	condition(s)
15	2	11	2	0	0	dataitem(s)
18	10	8	0	0	0	datatype(s)
0	0	0	0	0	0	state(s)

```
Deficiencies:          0
Inconsistencies:       0
```

Figure 22-3. T Software Description Metrics

```
T Design Rule Verification  Version 3.0
Copyright (C) 1987-1992 Programming Environments, Inc.
================================================================================
                                Translation
================================================================================
    1  /* T Design Rule Generation  Version 3.0
    2  ** Copyright (C) 1987-1992 Programming Environments, Inc.
    3  */
    4
    5  T_Packet                          tpacket
    6     s_packet                       Adalex_1
    7     .                               .
    8
    9  Local                             context_and_lexicon .
   10
   11  Local                             left_parenthesis .
   12
   13  Local                             right_parenthesis .
   14
   15  CombinationRule                   CR0001
   16     action                         Adalex;
   17     singular                       context_and_lexicon,
   18                                    identifier,
   19                                    letter,
   20                                    digit,
   21                                    letter_or_digit,
   22                                    integer,
   23                                    decimal_literal,
   24                                    operator_symbol,
   25                                    left_parenthesis,
   26                                    right_parenthesis,
   27                                    combining_option;
   28     .
   29
   30  SelectionRule                     SR0001
   31     datatype                       identifier_t;
   32     reference                      TBD;
   33     valid                          as_specified
   34                               with  function, boundary, debug;
   35     .
   36
   37  SelectionRule                     SR0002
   38     datatype                       letter_t;
   39     reference                      TBD;
   40     valid                          as_specified
   41                               with  function, boundary, debug;
   42     .
   43
   44  SelectionRule                     SR0003
   45     datatype                       digit_t;
   46     reference                      TBD;
```

Figure 22-4. T Design Rule Verification

```
47      valid                          as_specified
48                                with   function, boundary, debug;
49      .
50
51    SelectionRule                    SR0004
52      datatype                       letter_or_digit_t;
53      reference                      TBD;
54      valid                          as_specified
55                                with   function, boundary, debug;
56      .
57
58    SelectionRule                    SR0005
59      datatype                       integer_t;
60      reference                      TBD;
61      valid                          as_specified
62                                with   function, boundary, debug;
63      .
64
65    SelectionRule                    SR0006
66      datatype                       decimal_literal_t;
67      reference                      TBD;
68      valid                          as_specified
69                                with   function, boundary, debug;
70      .
71
72    SelectionRule                    SR0007
73      datatype                       operator_symbol_t;
74      reference                      TBD;
75      valid                          as_specified
76                                with   function, boundary, debug;
77      .
78
79    SelectionRule                    SR0008
80      datatype                       combining_option_t;
81      reference                      TBD;
82      valid                          as_specified
83                                with   function, boundary, debug;
84      .
85
86                                     pe_mark

 - finished translation with  13  recognizable TDRL sentences out of  13

================================================================================
                                Interpretation
================================================================================

s_packet: Adalex_1
unitdate: Fri Oct  9 13:13:49 1992
t_packet: tpacket
casedate: Fri Oct  9 13:17:42 1992

 - saving rules in test design data base
```

Figure 22-4 continued: T Design Rule Verification

```
T Test Catalog  Version 3.0
Copyright (C) 1987-1992 Programming Environments, Inc.

s_packet: Adalex_1
unitdate: Fri Oct  9 13:13:49 1992
t_packet: tpacket
casedate: Fri Oct  9 13:17:42 1992

Case     Purpose ( + exercises action, - fails to exercise action)

0001  +  action  Adalex
         state  <unspecified>
         dataitem  all at reference

0002  +  action  Adalex
         state  <unspecified>
         dataitem  all at low boundary

0003  +  action  Adalex
         state  <unspecified>
         dataitem  all at high boundary

0004  +  action  Adalex
         state  <unspecified>
         dataitem  identifier  (valid as_specified low_bound)

0005  +  action  Adalex
         state  <unspecified>
         dataitem  identifier  (valid as_specified high_bound)

0006  +  action  Adalex
         state  <unspecified>
         dataitem  identifier  (valid as_specified comp[2]_reference)

0007  +  action  Adalex
         state  <unspecified>
         dataitem  identifier  (valid as_specified comp[2]_reference)

0008  +  action  Adalex
         state  <unspecified>
         dataitem  identifier  (valid as_specified comp[2]_reference)
                                  ...

0095  +  action  Adalex
         state  <unspecified>
         dataitem  combining_option  (invalid out_of_type out_of_type_3)

 - saving cases in test design data base
```

Figure 22-5. T Test Catalog

```
T Sample Generation  Version 3.0
Copyright (C) 1987-1992 Programming Environments, Inc.

s_packet: Adalex_1
unitdate: Fri Oct  9 13:13:49 1992
t_packet: tpacket
casedate: Fri Oct  9 13:17:42 1992

combining_option
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     as_specified   reference            "is separate"
   [   2] valid     as_specified   low_bound            "is copied"
   [   3] valid     as_specified   high_bound           "is generic"
   [   4] invalid   not_in_list    not_in_list          "<not_in_list>"
   [   5] invalid   out_of_type    out_of_type_1        9
   [   6] invalid   out_of_type    out_of_type_2        9.9
   [   7] invalid   out_of_type    out_of_type_3        'a'

context_and_lexicon
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     placeholder    reference            "This ishould be the input cont

decimal_literal
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     as_specified   reference            "012.01"
   [   2] valid     as_specified   low_bound            "3."
   [   3] valid     as_specified   high_bound           "4567890123456789.23456789012345
   [   4] valid     as_specified   comp[2]_reference    "0.89"
   [   5] valid     as_specified   comp[2]_reference    "12.01"
   [   6] valid     as_specified   comp[2]_reference    "345678901234567.23"
   [   7] valid     as_specified   comp[2]_reference    "8901234567890123.45"
   [   8] valid     as_specified   comp[2]_low_bound    "456."
   [   9] valid     as_specified   comp[2]_low_debug    "789.6"
   [  10] valid     as_specified   comp[2]_high_debug   "012.7890123456789O1"
   [  11] valid     as_specified   comp[2]_high_bound   "345.23456578901234567"
   [  12] invalid   not_in_list    comp[0]_dropped      ".89"
   [  13] invalid   not_in_list    comp[1]_dropped      "67801"
   [  14] invalid   not_in_list    comp[0]_below_bounds ".23"
   [  15] invalid   not_in_list    comp[0]_above_bounds "90123456789012345.45"
   [  16] invalid   not_in_list    comp[1]_below_bounds "67867"
   [  17] invalid   not_in_list    comp[1]_above_bounds "901..89"
   [  18] invalid   not_in_list    comp[2]_above_bounds "234.01234567890123456"
   [  19] invalid   not_in_list    comp[0]_wrong        " !".78"
   [  20] invalid   not_in_list    comp[1]_wrong        "567 90"
   [  21] invalid   not_in_list    comp[2]_wrong        "890. !"#$%&'()*+,-./"
   [  22] invalid   out_of_type    out_of_type_1        9
   [  23] invalid   out_of_type    out_of_type_2        9.9
   [  24] invalid   out_of_type    out_of_type_3        'a'
```

Figure 22-6.T Sample Generation

digit

Index	SubDomain	Equiv.Class	Label	Value
[1]	valid	as_specified	reference	'0'
[2]	valid	as_specified	low_bound	'1'
[3]	valid	as_specified	high_bound	'2'
[4]	invalid	not_in_list	comp[0]_dropped	''
[5]	invalid	not_in_list	comp[0]_wrong	' '
[6]	invalid	out_of_type	out_of_type_1	9
[7]	invalid	out_of_type	out_of_type_2	9.9
[8]	invalid	out_of_type	out_of_type_3	"<not_in_list>"

identifier

Index	SubDomain	Equiv.Class	Label	Value
[1]	valid	as_specified	reference	"ABC_012"
[2]	valid	as_specified	low_bound	"D_3"
[3]	valid	as_specified	high_bound	"EFGHIJKLMNOPQRSTUVWXYZabcdefghi
[4]	valid	as_specified	comp[2]_reference	"Q_678"
[5]	valid	as_specified	comp[2]_reference	"RS_9AB"
[6]	valid	as_specified	comp[2]_reference	"TUVWXYZabcdefghijklmnopqrstuvwx
[7]	valid	as_specified	comp[2]_reference	"efghijklmnopqrstuvwxyzABCDEFGHI
[8]	valid	as_specified	comp[2]_low_bound	"qrs_I"
[9]	valid	as_specified	comp[2]_low_debug	"tuv_JK"
[10]	valid	as_specified	comp[2]_high_debug	"wxy_LMNOPQRSTUVWXYZabcdefghijkl
[11]	valid	as_specified	comp[2]_high_bound	"zAB_MNOPQRSTUVWXYZabcdefghijklm
[12]	invalid	not_in_list	comp[0]_dropped	"_OPQ"
[13]	invalid	not_in_list	comp[1]_dropped	"CDERST"
[14]	invalid	not_in_list	comp[2]_dropped	"FGH_"
[15]	invalid	not_in_list	comp[0]_below_bounds	"_UVW"
[16]	invalid	not_in_list	comp[0]_above_bounds	"IJKLMNOPQRSTUVWXYZabcdefghijklm
[17]	invalid	not_in_list	comp[1]_below_bounds	"VWXabc"
[18]	invalid	not_in_list	comp[1]_above_bounds	"YZa__def"
[19]	invalid	not_in_list	comp[2]_below_bounds	"bcd_"
[20]	invalid	not_in_list	comp[2]_above_bounds	"efg_ghijklmnopqrstuvwxyz012345(
[21]	invalid	not_in_list	comp[0]_wrong	" !"_jkl"
[22]	invalid	not_in_list	comp[1]_wrong	"hij mno"
[23]	invalid	not_in_list	comp[2]_wrong	"klm_ |""
[24]	invalid	out_of_type	out_of_type_1	9
[25]	invalid	out_of_type	out_of_type_2	9.9
[26]	invalid	out_of_type	out_of_type_3	'a'

integer

Index	SubDomain	Equiv.Class	Label	Value
[1]	valid	as_specified	reference	'012'
[2]	valid	as_specified	low_bound	'3'
[3]	valid	as_specified	high_bound	'456789012345678901234567890123(
[4]	valid	as_specified	comp[0]_low_bound	'8'
[5]	valid	as_specified	comp[0]_low_debug	'90'

Figure 22-6 continued: T Sample Generation

```
   [   6] valid     as_specified   comp[0]_high_debug   '12345678901234567890123456789012'
   [   7] invalid   not_in_list    comp[0]_dropped      ''
   [   8] invalid   out_of_type    out_of_type_1        9
   [   9] invalid   out_of_type    out_of_type_2        9.9
   [  10] invalid   out_of_type    out_of_type_3        "<not_in_list>"

left_parenthesis
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     <none>         fixed_value          "("

letter
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     as_specified   reference            "A"
   [   2] valid     as_specified   low_bound            "B"
   [   3] valid     as_specified   high_bound           "C"
   [   4] invalid   not_in_list    comp[0]_dropped      ""
   [   5] invalid   not_in_list    comp[0]_above_bounds "DE"
   [   6] invalid   not_in_list    comp[0]_wrong        " "
   [   7] invalid   out_of_type    out_of_type_1        9
   [   8] invalid   out_of_type    out_of_type_2        9.9
   [   9] invalid   out_of_type    out_of_type_3        'a'

letter_or_digit
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     as_specified   reference            '0'
   [   2] valid     as_specified   low_bound            '1'
   [   3] valid     as_specified   high_bound           '2'
   [   4] invalid   not_in_list    comp[0]_dropped      ''
   [   5] invalid   not_in_list    comp[0]_wrong        ' '
   [   6] invalid   out_of_type    out_of_type_1        9
   [   7] invalid   out_of_type    out_of_type_2        9.9
   [   8] invalid   out_of_type    out_of_type_3        "<not_in_list>"

operator_symbol
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     as_specified   reference            "*"
   [   2] valid     as_specified   low_bound            "+"
   [   3] valid     as_specified   high_bound           "/"
   [   4] valid     as_specified   element_2            "-"
   [   5] invalid   not_in_list    not_in_list          "<not_in_list>"
   [   6] invalid   out_of_type    out_of_type_1        9
   [   7] invalid   out_of_type    out_of_type_2        9.9
   [   8] invalid   out_of_type    out_of_type_3        'a'

right_parenthesis
   Index  SubDomain Equiv.Class    Label                Value
   -----  --------- ------------   -------------------  --------------------
   [   1] valid     <none>         fixed_value          ")"

 - saving samples in test design data base
```

Figure 22-6 continued: T Sample Generation

```
T Test Case Definitions  Version 3.0
Copyright (C) 1987-1992 Programming Environments, Inc.

s_packet: Adalex_1
unitdate: Fri Oct  9 13:13:49 1992
t_packet: tpacket
casedate: Fri Oct  9 13:17:42 1992

################################################################################
INDEX               0001
CASENAME            01000001
EXERCISES           Adalex
IN STATE            <unspecified>
REASON              inputs all at reference values
INPUT DATA
Name --- Value
--------------------------------------------------------------------------------
context_and_lexicon
     --- "This ishould be the input context and lexicon description."
identifier
     --- "ABC_012"
letter
     --- "A"
digit
     --- '0'
letter_or_digit
     --- '0'
integer
     --- '012'
decimal_literal
     --- "012.01"
operator_symbol
     --- "*"
left_parenthesis
     --- "("
right_parenthesis
     --- ")"
combining_option
     --- "is separate"
--------------------------------------------------------------------------------
START BY            invocation
END BY              termination
OUTPUT DATA
Name --- Value
--------------------------------------------------------------------------------
a_new_lexical_analyzer
     --- "Ada source code for a scanner"
Standard_Error_File
     --- "A report of errors detected"
--------------------------------------------------------------------------------
TRANSITION          <none>
```

Figure 22-7. T Test Case Definitions

```
################################################################################
INDEX                0002
CASENAME             01000002
EXERCISES            Adalex
IN STATE             <unspecified>
REASON               inputs all at low boundary
INPUT DATA
Name --- Value
--------------------------------------------------------------------------------
context_and_lexicon
     --- "This ishould be the input context and lexicon description."
identifier
     --- "D_3"
letter
     --- "B"
digit
     --- '1'
letter_or_digit
     --- '1'
integer
     --- '3'
decimal_literal
     --- "3."
operator_symbol
     --- "+"
left_parenthesis
     --- "("
right_parenthesis
     --- ")"
combining_option
     --- "is copied"
--------------------------------------------------------------------------------
START BY             invocation
END BY               termination
OUTPUT DATA
Name --- Value
                                      ...
--------------------------------------------------------------------------------
START BY             invocation
END BY               termination
OUTPUT DATA
Name --- Value
--------------------------------------------------------------------------------
a_new_lexical_analyzer
     --- "Ada source code for a scanner"
Standard_Error_File
     --- "A report of errors detected"
--------------------------------------------------------------------------------
TRANSITION           <none>
```

Figure 22-7 continued: T Test Case Definitions

23. T-PLAN

T-PLAN is a test planning and modeling method with an associated PC-based tool. The T-PLAN method is documented in a series of manuals that deal with strategic test planning, resource organization, and management. These manuals include guidelines that help a user to structure and partition testing into manageable pieces dependent on a series of factors such as risk and budget. The associated tool supports planning, organizing, and documenting test activities. The following evaluation focuses solely on the T-PLAN tool.

Aimed primarily at functional testing at the system level, the T-PLAN tool can be used for testing activities throughout development. In addition to supporting the planning and documentation of test activities, it provides statistical analyses to monitor these activities. Change impact analysis identifies those parts of a system under test that are affected by a modification.

23.1 Tool Overview

T-PLAN is marketed by Software Quality Assurance, Ltd. in England. This company will examine a customer's testing requirements to develop an implementation plan for T-PLAN installation. This service can include T-PLAN customization through the development of appropriate test models and data entry templates. Software Quality Assurance also provides strategic test planning consultancy and independent system testing, as well as training and seminars.

The tool has been available since 1989 and has over 120 users. It runs on an IBM PC, or compatible, under either DOS (release 3.1 onwards) or Unix. T-PLAN employs a fourth generation language and associated relational database. It can be networked on all major PC networks to enable a team of people to design, document, and review testing details. At the time of evaluation, T-PLAN prices started at £9,500. The evaluation was performed on a demonstration copy of T-PLAN version 2.0. This demonstration software is fully functional, except (1) only a limited number of records can be added to the supplied test data base, and (2) test data input and output templates cannot be customized.

T-PLAN allows the user to define the underlying test model, although the model defined by Software Quality Assurance can be used for this purpose. The structure of the resulting test model is recorded in the T-PLAN Test Dictionary. Consequently, with T-PLAN, the test process starts with establishing the structure of the underlying test dictio-

nary. The desired structure is typically determined through a modeling activity that, based on documentation such as the functional requirements description, specifies the expected behavior of the system under pre-determined conditions. It is specified in terms of functions, inputs, and outputs, called *test entities*. Specifically, the following object types are recognized:

- Test Specification. The highest level entity that defines the overall test plan.
- System Function. A condition to be tested, given at the level where individual test conditions can be identified for each function. Where necessary, system functions can be equated to system properties such as performance, recovery, or stress.
- Source System Input. Content and format of the input data required for testing.
- System Output/Data Profile. The contents and format of expected output data.
- System Program. An optional entity that allows storing program references and cross-referencing these to functions and files so that the affect of a program change on testing can be assessed.
- System File. An optional entity that allows storing file references and cross-referencing these to functions and programs so that the affect of a program change on files can be assessed.
- Service Queries. A record of the complete history of a change.

Once the dictionary structure has been established, Test Conditions are entered for the identified functions. These provide descriptions of the functional conditions that are required to be tested. Usually, they drive the design of test inputs and expected outputs. Test Conditions are captured with a unique reference and structured into Test Sets via a Function Reference; further grouping of Valid and Invalid categories can be assigned. Conditions relating to particular releases or versions of a system can also be grouped together. Test Conditions are cross-referenced to test inputs and expected results. Additionally, common Test Conditions that are to be centrally held and reused can be defined.

Test input data is created via user-tailorable templates. These are designed to match source system inputs (usually screens), thereby giving the feel and look of using the actual system. Special templates for "No Screen Data" testing can be used for scripting and allow script narrative to be cross-referenced to Test Conditions. Scripting can be combined with input templates as required.

The user also defines templates for expected output data, called Output Data Profiles. These data profiles are designed to match system outputs or to give a logical pointer to where output data is expected. Each represents a view of a record, file, or report of a specific data entity in a given time-frame. Thus, the history of a data entity can be recorded and tracked through the entire test plan. A special "No Profile Data" template is available for

capturing expected results. This template allows the user to give a narrative expected result "Check List" and cross-reference it to Test Conditions. As before, this type of scripting can be combined with the output data profile templates as required.

The dictionary employs cross-referencing at both the test entity level and the data level. Various static analysis functions make cross-reference listings available to the user. This cross-referencing enables T-PLAN to analyze the impacts of changes on Systems Functions, Inputs and/or Outputs. By identifying areas of the system that are directly affected by a change, and areas functionally dependent on the area being changed, T-PLAN helps to identify regression testing requirements.

The test dictionary itself consists of test specifications that contain the information required to test and check test results for a given part of the system. A test specification can contain one or more *test paths*, where a test path is a collection of tests that form a test run. Test paths can be thought of as a timeframe in which a particular set of tests must be run. They can, in turn, be grouped into *test cycles* to define a series of dependent test timeframes and/or test specifications. A log of test data and dated records of test events are subsequently stored against a test specification to allow a history of testing activities to be maintained. By mapping test specifications to software components, the dictionary provides for traceability of test data to the software under test. If required, test data may be linked across test specifications and its history tracked via Data Profiles. Traceability is also provided at the entity level, that is, between test specifications and functions, inputs, and outputs. Test schedule activities are defined in terms of a system, phase, package, and, optionally, software build parameters. They are separated under three categories: test preparation, testing, and regression testing, each of which has an associated review process. In each case, the user can specify who is responsible for the activity, an estimate of the number of days required, the actual days completed so far, and an estimate of days outstanding. This data is used to produce test progress reports. These reports include figures of, for example, percentage completed against schedule. The management reports provide information at the system, build, or package and phase levels. A Test Management Summary reports on the status of testing with respect to test paths.

Changes, and change control information, can also be recorded against test specifications. The change control management system utilizes "Service Queries." The user can record queries and errors, whether or not they sponsor a change, as service queries. This enables tracking the complete history of a change, including description, prioritization, estimates, actual and outstanding effort, query or error classification, and software library re-

lease information. Management reports provide for monitoring the progress of changes. For example, reports analyze the totals of errors or queries by classification, the frequencies or errors and frequency of clear-up, as well as the percentage completed and outstanding effort to complete changes. This type of information can be used to check test progress and assist in planning future projects.

23.2 Observations

Ease of use. The tool provides a menu driven interface, where the user uses the keyboard to make selections and input data. Database-type search and display operations are provided for viewing the test dictionary contents. There is no graphics capability. The online help facility provides a description, derivation formula, and cross reference to user manuals.

Templates are used throughout to facilitate data input. Since the user may customize existing templates, and create new templates, this provides some tailorability to the tool. To keep stored test data in line with the templates, when a template is changed, all test data stored for the template is automatically reformatted to match the new template. In certain circumstances, data captured via input templates can be exported in external file formats to aid in setting up file data needed for testing. The file formats available include DIF, dBase, Lotus 1-2-3. Conversely, data in these formats can be imported into T-PLAN.

Another helpful feature is the ability to create *central data profiles*. These are centrally held tables of data that can be accessed from input and output templates. This not only reduces the need to rekey repeated data but reduces the possibility of discrepancies between repeated pieces of data. The tool comes with one centrally held file already set up to hold a copy of Error Messages and Codes. (System error messages file data can be imported directly into the tool.)

Documentation. Only the documentation for the demonstration version of the tool was supplied. This was adequate for its purpose. The full version of T-PLAN is accompanied by four volumes relating to test management, strategy, test modeling, and technical issues. Installation was straightforward.

Problems encountered. The tool operated as described in the documentation.

23.3 Recent Changes and Planned Additions

An interface from T-PLAN to Direct Technology Ltd. Automator-QA is now available. This allows test input data stored in T-PLAN to be "played" directly into the software under test via Automator test scripts. An interface to automatically feed test log information back into T-PLAN is under development.

23.4 Sample Figures

Figures 23-1 through 23-17 provide sample outputs from T-PLAN.

```
                FUNCTIONAL CONDITION LIST   AS AT 04/01/80  VERSION 1.1
          FUNCTION REF      : FXI         FOR TEST SPEC REF: FA2
          FUNCTION NAME     : Foreign Exchange Input
          SYSTEM            : IBS
          OVERALL FUNCTION  : FX On-line Deal Processing

               RELEASE NO. 1.0          INCLUDE/EXCLUDE I
                            INVALID/VALID : I

TEST                CONDITION                          PATH      DEV DOC
 NO.                                                    NO.       REFS

 01 A   Deal Type not completed                         01      SCSD2.1
 01 B   Deal Type not valid code                        01      SCSD2.1
 02 B   Deal Date later than value date                 01      SCSD2.3
 02 C   Forward valued Deal Date before value date     01      SCSD2.3
                                    ...

                FUNCTIONAL CONDITION LIST   AS AT 04/01/80  VERSION 1.1
          FUNCTION REF      : FX2         FOR TEST SPEC REF: FA2
          FUNCTION NAME     : Foreign Exchange 2nd Authorise
          SYSTEM            : IBS
          OVERALL FUNCTION  : FX On-line Deal Processing

               RELEASE NO. 1.0          INCLUDE/EXCLUDE I
                            INVALID/VALID : I

TEST                CONDITION                          PATH       DEV DOC
 NO.                                                    NO.        REFS

 50     Attempt to 2nd Authorise a Deal that           05      SCSD2.3
        has not been 1st Authorised
 51     Attempt to 2nd Authorise a Deal without        05      SCSD2.3
        appropriate security level
 52     Attempt to 2nd Authorise a Deal that                   SCSD2.3
        has already been 2nd Authorized
 53     Attempt to 2nd Authorise a Deal that           05      SCSD2.3
                                    ...
 16 J   Interest Arbitrage Deal (Deal Type FA)                 SCSD2.3

              PERCENTAGE OF TESTS TO BE INCLUDED    84.00%
                                    ...
```

Figure 23-1. T-PLAN Test Model Functional Condition List Report

```
                    +-------------------------+
                    |   INPUT REFERENCE  EIN  |
                    +-------------------------+
                 +-------------------------------------+
                 | FUNCTION KEY : ENTER                |
                 +-------------------------------------+
          +-----------------------------------------------------+
          | TEST SPEC REF  FIN     PATH 01     SEQUENCE NO. 0010 |
          +-----------------------------------------------------+

DLIN X                                FX DEAL INPUT          DDMMYY  TERM 0A1P
           DEAL TYPE VN DEAL DATE 271290
         COUNTERPARTY 00109
   PURCHASED CURRENCY DEM
        SOLD CURRENCY NOK   AMOUNT 225,000
           CROSS RATE 4.444444
         STERLING RATE 2.934
           VALUE DATE 271290 OPTION FROM          TO
  CONFIRMATIOM METHOD S          PAYMENT METHOD N
       DEALING METHOD 10 COVER PAYMENT REQUIRED N
       SWAP BASE RATE                CCY IF NOT US$
A-TYPE DEAL SPOT RATE                 FLAT CURRENCY
  OUR RECEIVING AGENT C 00109HN0101
     OUR PAYIMG AGENT A 00109DB0107
THEIR RECEIVING AGENT
  BENEFICIARY ACCOUNT
          FREE FORMAT


DL   X       SUPPLEMENTARY PAYMENT DETAILS AND CHARGES          TERM 0A1P
A/C REF.  CODE NARRATIVE            CCY D/C AMOUNT          ERRORS/EXPANSIONS
                      SUPPLEMENTARY PAYMENT DETAILS
50   :ORDERING CUSTOMER
 NAM                                NAM
 STR                                PLC
57   :"ACCOUNT WITH" BANK            ACT/
 NAM                                NAM
 STR                                PLC
59   :BENEFICIARY CUSTOMER           ACT/
 NAM                                NAM
 STR                                PLC
70   :DETAILS OF PAYMENT
sp .5
71   :DETAILS OF CHARGES     DIRECT PAYMENT METHOD

                             ERROR MESSAGE
+-------------------------------------------------------------------+
|                        TEST PLAN NOTES/SCRIPT                     |
|  Note - Supplementary Payments screen should not appear           |
+-------------------------------------------------------------------+
                                 ...
```

Figure 23-2. T-PLAN Test Model Sample Print for Input Ref

```
          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
              TEST SPECIFICATION / INPUT REFERENCE MATRIX

TEST SPEC REFERENCE     : FIN

INPUT REFERENCE     : DMN  Deal Menu
INPUT REFERENCE     : EEN  Foreign Exchange Data Enquiry
INPUT REFERENCE     : EIN  Foreign Exchange Deal Input


T-PLAN TEST DICTIONARY   AS AT 04/01/80  VERSION 1.1
TEST SPECIFICATION / OUTPUT REFERENCE MATRIX

TEST SPEC REFERENCE : FIN  Foreign Exchange Deal Input/Summary Reports

OUTPUT REFERENCE     : DEL Deal
OUTPUT REFERENCE     : FBS FX Batch Summary Report
OUTPUT REFERENCE     : FIS FX Deal Input Summary Report


        INDEX OF OUTPUT REFERENCES    AS AT 04/01/80  VERSION 1.1
=============================================================================
  INPUT                      INPUT
   REF                       NAME
-----------------------------------------------------------------------------
   DMN             Deal Menu
   EA1             Foreign Exchange Deal 1st Authorise
   EA2             Foreign Exchange Deal 2nd Authorise
   EAM             Foreign Exchange Deal Amend
   EDE             Foreign Exchange Deal Delete
   ENE             Foreign Exchange Deal Enquiry
                              ...
   MIN             Money Movement Input
   STA             General Deal Status Enquiry Screen
=============================================================================


        INDEX OF OUTPUT REFERENCES    AS AT 04/01/80  VERSION 1.1
=============================================================================
  OUTPUT                    OUTPUT
   REF                       NAME
-----------------------------------------------------------------------------
   APL               Monthly P&L Report
   ATP               Trial Balance Report
   BEF               Bulked Entry File
   CAB               C&N Average Daily Balance Summary
   CAC               Customer Acount
   CCY               Foreign Exchange Rates
                              ...
   SPA               System Parameter Data
   SWI               SWIFT Messages
=============================================================================
```

Figure 23-3. T-PLAN Test Model Input & Output References for Test Spec FIN

```
                    NO SCREEN DATA TESTING (NSD)

SOURCE INPUT REF : EIN  Foreign Exchange Deal Input
   TEST SPEC REF : FIN    PATH NUMBER : 01    SEQUENCY NO. 0060

CONDITION REF                TEST NOTES                          TESTED OK?

  FXI21A   Orderer Client - 32345
  FXI21B   Orderer Client - Non Swift &<
  FXI21G   Orderer Client 68213 - Account With Bank 99-00-88
  FXI21H   Account With Bank MADGGG02
  FXI21I   Account With Bank 02356
  FXI21J   Account With Bank 80197
  FXI21K   Account With Bank Non Swift &@
                                    ...
  FXI21P   Details of Payment spaces - Details of charges BEF
```

Figure 23-4. T-PLAN Test Model No Screen Data Testing for FIN

```
OUTPUT DATA PROFILE NAME : DEAL DATA PROFILE
                           MAJOR  SUB TIME          DATA PROFILE
                           DP KEY KEY FRAME (PATH)     REF
OUTPUT DATA PROFILE REF  : DEL 003 00 01             DEL0030001
TEST SPEC REF            : FIN

                                NOTES
Deal should be deleted after batch overnight run

DEAL NUMBER     Syst gen       DEAL TYPE SN DEAL DATE 271290
           COUNTERPARTY 00203
     PURCHASED CURRENCY GBP  AMOUNT 10,000,000.00
          SOLD CURRENCY USD
             CROSS RATE 1.735
          STERLING RATE 1.75
             VALUE DATE 311290 OPTION FROM        TO
   CONFIRMATION METHOD S           PAYMENT METHOD S
        DEALING METHOD 10 COVER PAYMENT IF REQUIRED N
        SWAP BASE RATE              CCY IF NOT US$
  A-TYPE DEAL SPOT RATE              FLAT CURRENCY
    OUR RECEIVING AGENT C 12973HN0505
       OUR PAYING AGENT  22734HN0202
  THEIR RECEIVING AGENT  45798
    BENEFICIARY ACCOUNT
INHIBIT ADVICE TO RECEIVE?
71   :DETAILS OF CHARGES       DIRECT PAYMENT METHOD

          DEAL STATUS 01   CONF STATUS 01   PAYMENT STATUS 01
```

Figure 23-5. T-PLAN Test Model Output Print for FIN

```
        SPECIFICATION INSTRUCTIONS   AS AT 04/01/80  VERSION 1.1
    TEST SPEC REF : FIN  Foreign Exchange Deal Input/Summary Reports
              PAGE NUMBER : 01      AUTHOR : THEO CCUPIER

           TEST SPECIFICATION PREREQUISITES AND INSTRUCTIONS
           ------------------------------------------------

This Test Specification tests the following functionality in the IBM
system:-
        Foreign Exchange Input
        Foreign Exchange Enquiry (Part)
        FX Batch Summary (Part)
        FX Deal Input Summary (Part)

Note - Only part of some of the above functions are tested in this
Specification

The Test Spec has Input for the following days in the cycle:-
        01, 05

The Test Spec has output to be checked for the following days
        01, 05

Before commencing testing for PATH 1, ensure that the following test
environment is in place:-
        - System Test Libraries correctly loaded
        - System Test Base Data files correctly loaded
           - Customer and Account files
           - Rates files
           - User & Password files
           - System parameter file data for Day 1


       TEST PATH SUMMARY REPORT   AS AT 04/01/80  VERSION 1.1
    TEST SPEC REF :FIN Foreign Exchange Deal Input/summary Reports
                          PATH NUMBER 05

Logon to the system with User-id & Password X1AA
Enter Script Input from Path 05
Ensure that a note is made of all Deal Numbers allocated
After Entering all data and PRIOR to running Day 5 Batch Reports
Print DEAL file and check against data profiles

After running Batch Reports for Path 05 Check that Deals are correctly
deleted from DEAL file as noted on data profiles


            TEST SPECIFICATION TESTING LOG BY PATH
    TEST SPEC REF :FIN Foreign Exchange Deal Input/summary Reports
                  PATH NUMBER      TEST DATE
                     01     FIN01  19/09/91

       TESTER : GAH                        RETEST REQUIRED : yes

                              COMMENTS

Service Query 1 raised
```

Figure 23-6. T-PLAN Test Model Test Specification Information for FIN

Cycle (02)/Path Overview Report

```
TEST CYCLE/PATH OVERVIEW  AS AT 04/01/80  VERSION 1.1

                    CYCLE /PATH NUMBER    :  02
                    RUN SEQUENCE NO.      :  001

TEST SPEC REF : MM1  Money Movement 1st Authorisation
                              COMMENTS
Initial MM 1st Auth testing
set-up of deals & 1st auth

                              ...

TEST CYCLE/PATH OVERVIEW  AS AT 04/01/80  VERSION 1.1

CYCLE /PATH NUMBER    :  02
RUN SEQUENCE NO.      :  003

TEST SPEC REF : FA2  Foreign Exchange Deal 2nd Authorisation
                              COMMENTS
'Base Data" setup for FX2 Testing
```

Cycle (02)/Path Summary Report

```
        000   BAS Base data set-up
        001   FF1 Forex Fixed Deal 1st Authorisation
        001   MM1 Money Movement 1st Authorisation
        001   FA1 Foreign Exchange Deal 1st Authorisation
        001   DI1 Discounted Items 1st Authorisation
        002   DEE general Deal Enquiries
        003   FAD Foreign Exchange Deal Amend/Delete/Write-Off
        003   FA2 Foreign Exchange Deal 2nd Authorisation
```

Test Specification Input/Condition Cross Reference Report for FIN

```
                              FXI12A
                              FXI12B
                              FXI12C
                              FXI13A
                              FXI13B
                              FXI14A
                              FXI14B
                                ...
                              FXI20A

TEST SPECIFICATION INPUT BY PATH SUMMARY    AS AT 04/01/080  VERSION 1.1
TEST SPEC REF : FIN Foreign Exchange Deal Input/Summary Reports
```

Figure 23-6 continued: T-PLAN Test Model Test Specification Information for FIN

```
PATH NUMBER 01          FUNCTIONAL          INPUT
                        CONDITION REF        REF

  SEQUENCE NO. 0005     NOXREF               DMN
  SEQUENCE NO. 0010     FXI01A               EIN
  SEQUENCE NO. 0020     FXI01B               NSD
                                FXI02B
                                FXI02C
                                FXI02D
                          ...
  SEQUENCE NO. 0110     FXE11                EEN
                        FXE17
```

PATH NUMBER 01

TESTED BY *	DATE *	CHECKED BY *	DATE
*	*	*	
*	*	*	

```
PATH NUMBER 05          FUNCTIONAL          INPUT
                        CONDITION REF        REF

  SEQUENCE NO. 0005                          DMN
                        NOXREF
```

PATH NUMBER 05

TESTED BY *	DATE *	CHECKED BY *	DATE
*	*	*	
*	*	*	

Test Specification Exception Report

```
    ...
TEST SPEC REF : FIN Foreign Exchange Deal Input/Summary Reports

RELEASE NO. 1.0         INVALID/VALID V
FUNCTIONAL CONDITION REFERENCE
                        FXI24I
                        FXI24J
                        FXI15O
                        FXI15P
                        FXI15R
                         ...
                        FXI24H
FUNCTIONAL CONDITIONS NOT CROSS REFERENCED   AS AT 04/01/80  VERSION 1.1
    ...
```

Figure 23-6 continued: T-PLAN Test Model Test Specification Information for FIN

```
        INDEX OF FUNCTION REFERENCES   AS AT 04/01/80  VERSION 1.1
=============================================================================
     FUNCTION                        FUNCTION NAME
       REF
-----------------------------------------------------------------------------
          SYSTEM : IBIS
  OVERALL FUNCTION : Call & Notice - Money Movement
         MM1    Money Movement 1st Authorise
         MM2    Money Movement 2nd Authorise
         MMA    Money Movements Amend
         MMD    Money Movements Delete
         MME    Money Movements Enquiry
         MMI    Money Movements Input
                                    ...
=============================================================================
     FUNCTION                        FUNCTION NAME
       REF
-----------------------------------------------------------------------------
          SYSTEM : IBIS
  OVERALL FUNCTION : Position Enquiries
         PCU    Customer Positions
         PCY    Currency Positions
         PDL    Dealer Positions

Money Movement Enquiry
        INDEX OF INPUT REFERENCES   AS AT 04/01/80  VERSION 1.1
=============================================================================
  INPUT                          INPUT
   REF                           NAME
-----------------------------------------------------------------------------
    DMA        Deal Menu
    EA1        Foreign Exchange Deal 1st Authorise
    EA2        Foreign Exchange Deal 2nd Authorise
    EAM        Foreign Exchange Deal Amend
    EDE        Foreign Exchange Deal Delete
                                    ...
=============================================================================

Money Movement Enquiry
        INDEX OF OUTPUT REFERENCES   AS AT 04/01/80  VERSION 1.1
=============================================================================
  OUTPUT                         OUTPUT
   REF                           NAME
-----------------------------------------------------------------------------
    CIR        C&N Interest Rate Change Notification
    CNS        C&N Daily Summary Reports
    CUS        Customer
    DDD        DBR Daily Detail Report
    DEL        Deal
                                    ...
=============================================================================
```

Figure 23-7. T-PLAN Test Dictionary Function, Input, Output Reference Index

```
          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
             TEST SPECIFICATION / FUNCTION REFERENCE MATRIX

TEST SPEC REFERENCE     : CIR  Call & Notice Interest Rate Change Notification
FUNCTION REFERENCE      : CIR  C&N Interest Rate Change Notification
                               ...

          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
             TEST SPECIFICATION / FUNCTION REFERENCE MATRIX

TEST SPEC REFERENCE     : FIN  Foreign Exchange Deal Input/Summary Reports

FUNCTION REFERENCE      : FBS  FX Batch Summary
FUNCTION REFERENCE      : FIS  FX Deal Input Summary
FUNCTION REFERENCE      : FXE  Foreign Exchange Enquiry
FUNCTION REFERENCE      : FXI  Foreign Exchange Input

          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
              TEST SPECIFICATION / INPUT REFERENCE MATRIX

TEST SPEC REFERENCE     : FIN  Foreign Exchange Deal Input/Summary Reports

INPUT REFERENCE         : DMN  Deal Menu
INPUT REFERENCE         : EA1 Foreign Exchange Deal 1st Authorise
INPUT REFERENCE         : EA2 Foreign Exchange Deal 2nd Authorise
INPUT REFERENCE         : EAM Foreign Exchange Deal Amend
INPUT REFERENCE         : EEN Foreign Exchange Deal Enquiry
INPUT REFERENCE         : EIN Foreign Exchange Deal Input

          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
              TEST SPECIFICATION / OUTPUT REFERENCE MATRIX

TEST SPEC REFERENCE     : FIN  Foreign Exchange Deal Input/Summary Reports

OUTPUT REFERENCE        : DEL  Deal
OUTPUT REFERENCE        : FBS  FX Batch Summary Report
OUTPUT REFERENCE        : FIS  FX Deal Input Summary Report
```

Figure 23-8. T-PLAN Test Dictionary Functions, Inputs, Outputs Used in FIN

```
                                ...
CONDITIONS IMPACTING ON OUTPUT DATA PROFILES  AS AT 04/01/80  VERSION 1.1
                     ACROSS ALL TEST SPECIFICATIONS

                        DATA PROFILE KEY : DEL002
PATH       : 01
TEST SPEC : Foreign Exchange Deal Input/Summary Reports

         DATA PROFILE REF : DEL0020001   CONDITION REF  : FXI11
SOURCE INPUT KEY(SP
FIN010040EIN

                                ...
```

Figure 23-9. T-PLAN Test Dictionary Condition Impact on Data Profiles

```
          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
                 FUNCTION CHANGE IMPACT ANALYSIS REPORT

FUNCTION REF      : MME  Money Movement Enquiry
=============================================================================
TEST SPEC REF     : MM1  Money Movement 1st Authorise
TEST SPEC REF     : MM2  Money Movement 2nd Authorise
TEST SPEC REF     : MMA  Money Movement Amend/Delete/Write-off
TEST SPEC REF     : MMI  Money Movement Input/Summary Reports

          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
                   INPUT CHANGE IMPACT ANALYSIS REPORT

INPUT REF      : EIN  Foreign Exhange Deal Input
=============================================================================
TEST SPEC REF : DEE  General Deal Enquiries
TEST SPEC REF : FA1  Foreign Exchange Deal 1st Authorisation
TEST SPEC REF : FA2  Foreign Exchange Deal 2nd Authorisation
TEST SPEC REF : FAD  Foreign Exchange Deal Amend/Delete/Write-off
TEST SPEC REF : FIN  Foreign Exchange Deal Input/Summary Reports
TEST SPEC REF : PEN  Position Enquiries

          T-PLAN TEST DICTIONARY    AS AT 04/01/80  VERSION 1.1
                  OUTPUT CHANGE IMPACT ANALYSIS REPORT

OUTPUT REF      : FIS  FX Deal Input Summary Report
=============================================================================
TEST SPEC REF : FIN  Foreign Exchange Deal Input/Summary Reports
```

Figure 23-10. T-PLAN Test Dictionary Change Impact for Function MME, Input EIN, Output FIS

```
FF2 Forex Fixed Deal 2nd Authorisation
FFA Forex Fixed Deal Amend/Delete/Write-off
MM1 Money Movement 1st Authorisation
MM2 Money Movement 2nd Authorisation
MMI Money Movement Input/Summary Reports
MMA Money Movement Amend/Delete/Write-off
DII Discounted Items  InputSummary Reports
 ...
NWD Non-Working day
            TEST SPECIFICATION INDEX   AS AT 04/01/080  VERSION 1.1
=============================================================================
TEST           TEST SPEC                                    AUTHOR
SPEC            NAME
REF
-----------------------------------------------------------------------------
FSW Btach FX SWIFT Message Generation
PUP Batch Position Update/Report
```

Figure 23-11. T-PLAN Test Dictionary Test Specification Index

SERVICE QUERY NUMBER	AS AT 13/05/92 TIME OF PRINT
00002	13:44:27

SYSTEM : IBS PHASE/PACKAGE : 1.1 SOFTWARE BUILD : 01

DESCRIPTION

Should not the FX Supplementary Screen only be Presented if
The Program Spec does not make it clear

AREA RAISED BY PGM WHO RAISED BY HHG DATE RAISED 01/09/91
PRIORITY CODE BY 02/09/91

SERVICE QUERY COMMENTS

No mention is in the program spec regarding the criteria for
display of the FX supplementary details screen
HHG - PGM

Spec amended accordingly
CDT - DES

No action for STT - Design spec specifies this
HH - STT

SERVICE QUERY STATISTICS

DESIGN	0.10	DESIGN	0.10	DESIGN	0.00
PROG	0.00	PROG	0.00	PROG	0.00
SYS TEST	0.00	SYS TEST	0.00	SYS TEST	0.00
OTHER	0.00	OTHER	0.00	OTHER	0.00
TOTAL	0.00	TOTAL	0.00	TOTAL	0.00

SERVICE QUERY ACTION AUTHORISED BY FF
SERVICE QUERY CLASSIFICATION 03 MAJOR DOCUMENTATION ERROR

Figure 23-12. T-PLAN Test Management Service Query Report for SQ 00002

TEST SPECIFICATION/SERVICE QUERY LOG

TEST SPEC : FIN - Foreign Deal Input/Summary Reports

SQ No.	Raised for Test Spec ?	Date Logged	SPEC UPDATE DETAILS: Reqd ?	Updated By	Updated Date	Checked By	Checked Date	REGRESSION TEST: Regn. Test Reqd.?	Test By
00001	yes	19/09/91							
00002	no	01/09/91							

Figure 23-13. T-PLAN Test Management Test Spec/SQ Log for FIN

SERVICE QUERY FREQUENCY & CLEAR UP REPORT

Period		Service Queries	
From	To	Raised	Signed-off
10/08/91 -	17/08/91	0	0
18/08/91 -	25/08/91	0	0

SERVICE QUERY ANALYSIS BY CLASSIFICATION WITHIN SCHEDULE

SYSTEM	PACKAGE /PHASE	SOFTWARE BUILD	SERVICE QUERY CALSSIFICATION	QUANTITY
IBS				
	1.1			
		01		
			03- MAJOR DOCUMENTATION ERROR	1
			07- MAJOR PROGRAMMING ERROR	1
		TOTAL for Software Build 01		2
	TOTAL for Package/Phase 1.1			2
TOTAL for System IBS				2
OVERALL TOTAL				2

Figure 23-14. T-PLAN Test Management Service Query Reports

OVERALL PROGRESS AS AT 16/03/92

TASK CODE	TASK DESCRIPTION	[-------Totals------] ORIG	ACT	OUTST	Percent Complete
PHASE/PACKAGE 1.1	SOFTWARE BUILD 01				
DEE	General Deal Enquiries	13.75	5.00	8.25	37.74 %
FA1	Foreign Exchange Deal 1st Authorisa	10.50	5.75	5.00	52.38 %
FIN	Foreign Exchange Deal Input/Summary	13.75	11.00	3.75	72.73 %
	SOFTWARE BUILD TOTALS	37.50	21.75	17.00	54.67 %
PHASE/PACKAGE 1.1	SOFTWARE BUILD 02				
FA2	Foreign Exchange Deal 2nd Authorisa	15.75	2.00	13.75	12.70 %
	SOFTWARE BUILD TOTALS	15.75	2.00	13.75	12.70 %
	PHASE/PACKAGE TOTALS	53.25	23.75	30.75	42.25 %
	OVERALL TOTALS	53.25	23.75	30.75	42.25 %

Figure 23-15. T-PLAN Overall Progress for IBS

SYSTEM : IBIS PHASE/PACKAGE : 1.1 ACTIVE SERVICE QUERY INDEX AS AT 16/03/92 TIME OF PRINT 13:07:31

SQ NUMBER	DESCRIPTION	AREA RAISED/BY	PRIORITY BY	AUTH BY	SQ CLASS	DATE WITH	AREA WHO WITH WITH
00002	Should not the FX Supplementary Screen only be presented if 'Y' is request in the 'Supplementary Field Required' field. The Program Spec does not make it clear.	PGM HHG	P 02/09/91	FF	03	03/09/91	PGM HHG

NUMBER OF ACTIVE SERVICE QUERIES FOR PHASE/PACKAGE 1.1 : 1
NUMBER OF ACTIVE SERVICE QUERIES FOR SYSTEM IBIS : 1

SERVICE QUERY SCHEDULE STATISTICS

SYSTEM : IBS PACKAGE/PHASE : 1.1 Printed on:16/03/92 at:13:12:34

SOFTWAR BUILD	SQ NO.	ESTIMATES DESIGN	PROG	SYSTEM TESTING	OTHER	TOTAL	ACTUALS DESIGN	PROG	SYSTEM TESTING	OTHER	TOTAL	OUTSTANDING DESIGN	PROG	SYSTEM TESTING	OTHER	TOTAL	PERCENT COMPLETE
01																	
	00001	0.00	0.25	0.00	0.25	0.50	0.00	0.25	0.25	0.00	0.50	0.00	0.00	0.00	0.00	0.00	100.00%
	00002	0.10	0.00	0.00	0.00	0.10	0.10	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00	100.00%
BUILD TOTAL		0.10	0.25	0.00	0.25	0.60	0.10	0.25	0.25	0.00	0.60	0.00	0.00	0.00	0.00	0.00	100.00%
PACKAGE TOTAL		0.10	0.25	0.00	0.25	0.60	0.10	0.25	0.25	0.00	0.60	0.00	0.00	0.00	0.00	0.00	100.00%
SYSTEM TOTAL		0.10	0.25	0.00	0.25	0.60	0.10	0.25	0.25	0.00	0.60	0.00	0.00	0.00	0.00	0.00	100.00%
OVERALL TOTAL		0.10	0.25	0.00	0.25	0.60	0.10	0.25	0.25	0.00	0.60	0.00	0.00	0.00	0.00	0.00	100.00%

Figure 23-16. T-PLAN Test Management Service Query Reports

TESTING PROGRESS REPORTS FOR IBS

PREPARATION AND CHECKING AS AT 16/03/92

TASK CODE	TASK DESCRIPTION	[-----Preparation-----] WHO	ORIG	ACT	OUTST	[-----Prep Check------] WHO	ORIG	ACT	OUTST	[----Totals-----] ORIG	ACT	OUTST	Percent Complete
PHASE/PACKAGE 1.1	SOFTWARE BUILD 01												
DEE	General Deal Enquiries	CFC	5.00	2.50	2.50	BDB	1.00	0.50	0.50	6.00	3.00	3.00	50.00%
FA1	Foreign Exchange Deal 1st Authorisa	AGA	4.00	3.50	0.00	BDB	0.50	0.75	0.00	4.50	4.25	0.00	100.00%
FIN	Foreign Exchange Deal Input/Summary	AGA	5.00	6.00	0.00	BDB	1.00	1.50	0.00	6.00	7.50	0.00	100.00%
	SOFTWARE BUILD TOTALS		14.00	12.00	2.50		2.50	2.75	0.50	16.50	14.75	3.00	81.82%
PHASE/PACKAGE 1.1	SOFTWARE BUILD 02												
FA2	Foreign Exchange Deal 2nd Authorisa	CFC	6.50	2.00	4.50	BDB	1.00	0.00	1.00	7.50	2.00	5.50	26.67%
	SOFTWARE BUILD TOTALS		6.50	2.00	4.50	BDB	1.00	0.00	1.00	7.50	2.00	5.50	26.67%
	PHASE/PACKAGE TOTALS		20.50	14.00	7.00	BDB	3.50	2.75	1.50	24.00	16.75	8.50	64.58%
	OVERALL TOTALS		20.50	14.00	7.00	BDB	3.50	2.75	1.50	24.00	16.75	8.50	64.58%

Figure 23-17. T-PLAN Test Management Reports

TESTING PROGRESS REPORTS FOR IBS

TESTING AND CHECKING AS AT 16/03/92

TASK CODE	TASK DESCRIPTION	[------Testing-------] WHO	ORIG	ACT	OUTST	[------Test Check------] WHO	ORIG	ACT	OUTST	[---Totals-----] ORIG	ACT	OUTST	Percent Complete
PHASE/PACKAGE 1.1	SOFTWARE BUILD 01												
DEE	General Deal Enquiries	YY	5.00	2.00	3.00	HJG	0.50	0.00	0.50	5.50	2.00	3.50	36.36%
FA1	Foreign Exchange Deal 1st Authorisa	GAH	4.00	1.50	3.00	HJG	0.25	0.00	0.25	4.25	1.50	3.25	23.53%
FIN	Foreign Exchange Deal Input/Summary	GAH	5.00	3.50	1.00	BDB	0.50	0.00	0.50	5.50	3.50	1.50	72.73%
	SOFTWARE BUILD TOTALS		14.00	7.00	7.00		1.25	0.00	1.25	15.25	7.00	8.25	45.90%
PHASE/PACKAGE 1.1	SOFTWARE BUILD 02												
FA2	Foreign Exchange Deal 2nd Authorisa	YY	5.00	0.00	5.00	HJG	0.50	0.00	0.50	5.50	0.00	5.50	0.00%
	SOFTWARE BUILD TOTALS		5.00	0.00	5.00	HJG	0.50	0.00	0.50	5.50	0.00	5.50	0.00%
	PHASE/PACKAGE TOTALS		19.00	7.00	12.00	HJG	1.75	0.00	1.75	20.75	7.00	13.75	33.73%
	OVERALL TOTALS		19.00	7.00	12.00	HJG	1.75	0.00	1.75	20.75	7.00	13.75	33.73%

Figure 22-17 continued: T-PLAN Test Management Reports

TESTING PROGRESS REPORTS FOR IBS

REGRESSION AND CHECKING AS AT 16/03/92

TASK CODE	TASK DESCRIPTION	[------Regression------] WHO	ORIG	ACT	OUTST	[----Regress Check----] WHO	ORIG	ACT	OUTST	[----Totals-----] ORIG	ACT	OUTST	Percent Complete
PHASE/PACKAGE 1.1	SOFTWARE BUILD 01												
DEE	General Deal Enquiries	PPA	1.50	0.00	1.50	HJG	0.25	0.00	0.25	1.75	0.00	1.75	00.00%
FA1	Foreign Exchange Deal 1st Authorisa	UGH	1.50	0.00	1.50	HJG	0.25	0.00	0.25	1.75	0.00	1.75	00.00%
FIN	Foreign Exchange Deal Input/Summary	UGH	2.00	0.00	2.00	BDB	0.25	0.00	0.25	2.75	0.00	2.25	00.00%
	SOFTWARE BUILD TOTALS		5.00	0.00	5.00		0.75	0.00	0.75	5.75	0.00	5.75	00.0%
PHASE/PACKAGE 1.1	SOFTWARE BUILD 02												
FA2	Foreign Exchange Deal 2nd Authorisa	PPA	2.50	0.00	2.50	HJG	0.25	0.00	0.25	2.75	0.00	2.75	0.00%
	SOFTWARE BUILD TOTALS		2.50	0.00	2.50	HJG	0.25	0.00	0.25	2.75	0.00	2.75	0.00%
	PHASE/PACKAGE TOTALS		7.50	0.00	7.50	HJG	1.00	0.00	1.00	8.50	0.00	8.50	0.00%
	OVERALL TOTALS		7.50	0.00	7.50	HJG	1.00	0.00	1.00	8.50	0.00	8.50	0.00%

Figure 22-17 continued: T-PLAN Test Management Reports

24. TBGEN and TCMON

TBGEN generates test drivers that facilitate unit testing and bottom-up integration testing. The latest version of this tool includes the generation of stubs so that top-down integration testing is also supported. TBGEN's companion tool, TCMON, provides structural coverage and timing analysis.

24.1 Tool Overview

Until recently, TBGEN and TCMON were marketed by ICL Personal Systems, formerly Nokia Data Systems. They are now available from Testwell Oy. These tools have been commercially available since 1986 and 30 permanent multi-user licenses have been sold. Designed to be hardware architecture, operating system, and compiler independent, these tools are available for VAX/VMS, Sun-3/SunOS, PCs under MS-DOS and OS-2, and Rational machines. There are some minor difference between the versions available on different operating environments; for example, unlike the Sun-3/SunOS versions, the VAX/VMS tools do not allow escaping to the operating system command level. At the time of examination, TBGEN prices started at $2,850 and TCMON at $2,300. The versions examined were TBGEN Version 3.1 and TCMON Version 2.2 operating on a VAX/VMS platform.

24.1.1 TBGEN Overview

Using Ada program unit specifications, TBGEN generates a test driver and a command file for compiling and linking this test driver with the units under test. The user can control the size of the resulting testbed by specifying particular subprograms or program objects to be excluded. A log file automatically records pertinent information about testbed generation. The user executes the resulting testbed, interactively specifying the desired calling sequence and subprogram parameters, and observing the results. (Since the testbed takes standard input from the keyboard for interactive communication with the user, some difficulties may be encountered if a module under test also uses standard input.)

A powerful set of Ada-like testbed commands is provided. For example, testbed variables can be declared and their visibility directly controlled, and many of the entities declared in Ada specifications can be accessed. Additional commands display information based on current testbed settings and testbed status, or cause user inputs and testbed outputs to be copied to a trace file for later examination. Instead of using a testbed interactively, the

user can specify testbed inputs in the form of a script file. Scripts may be user developed or generated from a copy of previous testbed inputs. Conditional and iteration control structures, along with fixed and variable breakpoints, are provided for scripts. Assertions are provided for automatic checking of test results against expected results.

24.1.2 TCMON Overview

TCMON instruments the contents of user-selected files with statements that act as measurement probes. These probes provide for coverage analysis at the segment, condition, and subcondition levels. In addition to structural coverage, probes provide for segment execution counts and true/false counts of conditions and subconditions. They also provide timers that allow capturing execution times at the program unit level and the measurement of times between user-specified events. Each subprogram can be instrumented for different types of monitoring. A test monitor is generated. A command file for compiling the monitor and instrumented code and performing necessary linking is also generated, together with a log file providing information about instrumented files and units generated. The monitor supports a command-driven interface that provides the user with commands such as those required to reset all counters and timers, save and append measurement data, produce a profile listing, and run the instrumented program. Where necessary, this interface can be omitted by inserting TCMON commands in source files as special comments and generating a dummy monitor. Data generated by the instrumentation is recorded in a profile listing. This gives detailed information about counter and timer places and values, and a histogram of statement list execution counts is included. The profile listing also contains information that can be used to estimate the influence of instrumentation statements on measured time. The TCMON Postprocessor (TCPOST) processes the profile listing to generate summary reports at either the package or subprogram level.

Timers may include invalid data when two or more tasks call the same instrumented subprogram or are of the same instrumented task type. The same is true for recursive procedures. If this happens, the affected timers are flagged in the profile listing. Although generic procedures and packages can be instrumented, multiple instances are not distinguished. Also, when returning from a function, it is not possible for a timer within the function to record the time spent in the evaluation of the return expression. Exceptions, which are invisible to the instrumentor, may also distort timing results.

24.2 Observations

Ease of use. The user interacts with TBGEN and TCMON through command interfaces that are well supported with prompts to guide a user through necessary steps. Context-sensitive help is available, together with general descriptions on user-selected topics. Error messages are informative, though no specific help for resolving an error is provided; messages are written to both the display and the appropriate log file. When erroneous input is detected, execution of the current command is terminated and the rest of the current input line ignored. When a test script is being used in TBGEN, processing will continue with the next line. Command files are provided to relieve the user of some repetitive manual labor. Although the use of TCMON requires no special knowledge, the TBGEN command-interface requires some knowledge of Ada. All reports are well-structured and clear, with useful history-keeping information.

TBGEN is tailorable in several ways. The SPECIAL command implements environment or installation specific commands. Configuration parameters specified in a system file can be changed, essentially to modify default file names. A system file gives the specification for package STANDARD which can be modified to reflect some of the options available to Ada compilers. The template files used in generating testbed components can be changed.

Some aspects of TCMON can also be altered by modifying the template file used for generating auxiliary Ada units and the command file. This template file also contains the configuration parameters that can be changed to alter default values. The TCMON User's Manual provides suggestions for modifying the parent type for counter variables, measuring CPU time instead of default elapsed time, and including other cost functions.

Documentation and user support. The documentation is well-written and guides a user through using each tool. The vendor provided good support and answered all questions quickly and well.

Instrumentation overhead. TCMON is designed to minimize the introduction of unnecessary instrumentation. It not only allows the user to select the files whose contents are to be instrumented, but allows each file to be instrumented differently. TCMON also allows the user to select between SAFE or UNCHECKED modes for the segment counter. The vendor cites a 50% to 100% increase in code size for full structural instrumentation. For the Ada Lexical Analyzer Generator, full structural instrumentation of all units gave a size increase of 120%.

Ada restrictions. TBGEN accepts any valid Ada code. Expressions, however, are skipped with the result that the type of an array index cannot always be determined automatically and the user may be asked to supply this information. Tasks, task types, and dependent entities are ignored and cannot be accessed in testbeds directly. Similarly, testbeds do not provide the user with access to objects of limited type, functions with results of limited type, array objects with a constrained array definition, and constrained subtypes of a type with discriminants. TCMON may misinterpret overloaded operators returning boolean values when these are used in conditions.

Problems encountered. No significant problems were encountered during the examinations of these tools.

24.3 Recent Changes

TBGEN version 3.0 has been ported to Apollo/Domain environments. An upgrade, version 3.1, is available on a limited set of platforms. The notable enhancements included in the upgrade are a recording facility for user input to allow automatic repetition of interactive test sessions and a blockwise USE command.

In November 1991, TBGEN version 4.0 was released and is now available for all the previous environments, except Rational machines. This version introduces the generation of stubs to facilitate top-down testing. TBGEN is the only identified tool that provides this powerful and valuable capability.

TBGEN and TCMON are also available through DDC International as an integrated part of its CASE Toolbox product. The Sun/SPARC version of these tools is also available through DDC International.

Under certain circumstances both tools can be licensed in Ada source code form with connection ports to Gould, Apollo/Domain, and some NEC machines (Unix environments).

24.4 Sample Outputs

Figures 24-1 through 24-6 provide sample outputs from TBGEN and TCMON.

```
-- Script file : USR:[ADATEST.TBGEN]CALENDAR.REC;1
-- Created at  : 1991-08-15 10:37:14
-- Created by  : Test bed CAL_BED generated at 1991-08-15 09:00:03
SET TRACE FILE calendar.trc
DECLARE
  USE calendar
  moment        : time := clock
  current_year  : year
  current_month : month
  the_day       : day_num
  seconds       : day_dura
BEGIN
  split(moment, current_year, current_month, the_day, seconds)

  moment := time_of(current_year, current_month, 15, 0.0)
  DISPLAY day(moment)
  moment := add__1(moment, 86400.0)  -- add__1 equiv to "+"
  split(moment, current_year, current_month, the_day, seconds)
  ASSERT the_day = 16 AND seconds = 0.0

  now     : time := clock
  later   : time := clock
  ASSERT le__1(now, later) = true  -- le__1 equiv to "<="

  moment := time_of(1991, 2, 28, 0.0)
  ASSERT NOT EXCEPTION

  moment := time_of(1991, 2, 29, 0.0)
  ASSERT EXCEPTION(time_error)
END
SET TRACE CLOSED
SET RECORD CLOSED
```

Figure 24-1. TBGEN Record File

```
*******************************************************************************
*                        Softplan (R) Ada Tools                               *
*     TBGEN System Version 3.1, Copyright (C) 1990 Nokia Data Systems         *
*                        Test Bed Trace Listing                               *
*******************************************************************************
Test bed generated at 1991-08-15 09:00:03. Time is now 1991-08-15 10:37:22
CAL_BED> DECLARE
CAL_BED>   USE calendar
CAL_BED>   moment        : time := clock
CAL_BED>   current_year  : year
CAL_BED>   current_month : month
CAL_BED>   the_day       : day_num
CAL_BED>   seconds       : day_dura
CAL_BED> BEGIN
CAL_BED>   split(moment, current_year, current_month, the_day, seconds)
         YEAR (out) =  1991
         MONTH (out) =  8
         DAY (out) =  15
         SECONDS (out) =  38266.8500
CAL_BED>
CAL_BED>   moment := time_of(current_year, current_month, 15, 0.0)
CAL_BED>   DISPLAY day(moment)
          15
CAL_BED>   moment := add__1(moment, 86400.0)  -- add__1 equiv to "+"
CAL_BED>   split(moment, current_year, current_month, the_day, seconds)
         YEAR (out) =  1991
         MONTH (out) =  8
         DAY (out) =  16
         SECONDS (out) =  0.0000
CAL_BED>   ASSERT the_day = 16 AND seconds = 0.0
CAL_BED>
CAL_BED>   now     : time := clock
CAL_BED>   later   : time := clock
CAL_BED>   ASSERT le__1(now, later) = true  -- le__1 equiv to "<="
CAL_BED>
CAL_BED>   moment := time_of(1991, 2, 28, 0.0)
CAL_BED>   ASSERT NOT EXCEPTION
CAL_BED>
CAL_BED>   moment := time_of(1991, 2, 29, 0.0)
         *** exception CALENDAR.TIME_ERROR
CAL_BED>   ASSERT EXCEPTION(time_error)
CAL_BED> END
CAL_BED> SET TRACE CLOSED

         Trace closed at 1991-08-15 10:43:46
```

Figure 24-2. TBGEN Trace File

```
*******************************************************************************
*                          Softplan (R) Ada Tools                             *
*     TBGEN System Version 3.1, Copyright (C) 1990 Nokia Data Systems         *
*                      Test Bed Generation Log File                           *
*******************************************************************************
Licence identification of the generator:
                  ...
Test bed timestamp...: 1991-08-15 09:00:03

Test bed name.......: CAL_BED
Generated Ada files..: cal*.ada
Command file........: calCMD.COM

Analysed source files:
File: TBGENSYS.STD
File: calendar.spe
                  ...
===============================================================================
The symbol table :

package STANDARD/8001/ is
   type BOOLEAN/1/ is (
      FALSE,
      TRUE);
   type INTEGER/2/ is Integer_Type;
   type FLOAT/3/NoV/ is Float_Type;
   type CHARACTER/4/NoV/ is (
      <>);
   subtype NATURAL/5/NoV/ is INTEGER <2> ;
   -- Type Class  => Integer_Type
   subtype POSITIVE/6/NoV/ is INTEGER <2> ;
   -- Type Class  => Integer_Type
                  ...
      function ">="/2015/(
         LEFT : in     CALENDAR.TIME <11> ;
         RIGHT : in     CALENDAR.TIME <11> )
         return BOOLEAN <1> ;
         -- Alias Name => GE__1
      TIME_ERROR/6006/ : exception;

   end CALENDAR;
end STANDARD;
===============================================================================
Execution of the generator successfully ended at 1991-08-15 09:01:02
```

Figure 24-3. TBGEN Generated Log File

```
*****************************************************************
*  TCMON System Version 2.2, (C) Copyright 1987 by Softplan     *
*  Test Coverage Monitor / Program Bottleneck Finder            *
*  Execution profile listing                                    *
*****************************************************************
                                           Counters          Timers
                                          ===========  ======================
LINE   EXECUTION  LTBGEN/TCMON-  PLACE              START/  END/          AVERAGE
NO.    COUNTS     VEL  DESCRIPTION        TRUE      FALSE     TIME            TIME

Source file => [-.adalex2]ll_decls.ada                      Instr => (A,N,N,N)
Source file => ll_compile_dummy.ada                         Instr => (A,N,N,Y)
  23............... 1 proc LL_COMPILE
 120............... 2 func LLFIND
 124*****>>>>>      2 begin               198       0 ?
 127*****>>>>>>>    3 while_loop          898     763 ?
                      Condition           898      63
                                 ...
Source file => [-.adalex2]ll_sup_spec.ada                   Instr => (A,N,N,N)
Source file => ll_sup_body_mt.ada                           Instr => (A,Y,Y,Y)
  50............... 1 pack LL_SUPPORT               0                      0.0000
  97............... 2 func ALTERNATE               14         0.0000       0.0000
 170............... 2 func CHAR_RANGE               6         0.0013       0.0078
 175*****>          2 begin                 6       0 ?
 176                  TIMER_CHAR_RANGE              6         0.0013       0.0078
 178***             3 if_then               3       3
                      Condition             3       3
 181***             3 if_else               3       3
 183*****>>>>       4 for_loop             62      62
 188*****>          3 return                6       0
 204............... 3 proc COMPLETE_ALT             4         0.0039       0.0156
                                 ...
Median of nonzero counter values =  8
One asterisk ( * )               <=> 1
Number of counters                    = 539
Number of timers                      = 36
Number of instrumented (sub)conditions = 206
Minimum measurable time interval      =        0.0001
Estimated cost of one timer operation =        0.0006
Estimated cost of one counter operation =      0.0000
The instrumentation was done 1991-08-14 13:15:27
This listing was produced    1991-08-14 13:32:48

Data files:
    NAME =>sampleTIM.dat
                ...
```

Figure 24-4. TCMON Profile Execution Listing

```
*****************************************************************
*  TCMON System Version 2.2, (C) Copyright 1987 by Softplan     *
*  Test Coverage Monitor / Program Bottleneck Finder            *
*  Log of TCMON preprocessor execution                          *
*****************************************************************
Date and time     => 1991-08-14 13:15:27

Prefix            => SAMPLE
Generated files   => sample*.ada
Main procedure    => *not specified*
Code pattern file => PATTERNS.TCM
                  ...
Source => ll_sup_body_mt.ada
Target => sample_ll_sup_body_mt.ada
Instrumentation => ( INC_MODE        => UNCHECKED
                     COUNTERS        => ALL
                     AUTO_TIMERS     => YES
                     MANUAL_TIMERS   => YES
                     EXPAND_COMMANDS => YES )

  package body LL_SUPPORT on line 50
     function ALTERNATE on line 97
        function MERGE_RANGES on line 103
     function CHAR_RANGE on line 170
     --&& start timer_char_range on line 176 expanded
     --&& stop timer_char_range on line 187 expanded
     procedure COMPLETE_PAT on line 192
                  ...
          Summary Information
          *******************

Number of instrumented files      =     6
Number of compilation units       =     4
Number of body stubs              =     2
Number of subunits                =     2
Number of statement list counters =   539
Number of (sub)condition counters =   206
Number of timers                  =    36
Number of manual timer STARTs     =     2   ... all expanded
Number of manual timer STOPs      =     2   ... all expanded
Number of embedded commands       =     1   ... all expanded

Command file for compilation and linking => SAMPLECMD.COM

ERRORS:    0  WARNINGS:    0
```

Figure 24-5. TCMON Log File

```
*****************************************************************************
*  TCMON System Version 2.2, (C) Copyright 1987 by Softplan                 *
*  Test Coverage Summary Report                                             *
*****************************************************************************
PROGRAM                                  STM      COND      SUB      OVER
UNIT                                     LIST     CVRG      COND     ALL
                                         CVRG               CVRG     CVRG

Source file => ll_compile_dummy.ada                        Instr => (A,N,N,Y)
proc LL_COMPILE
 func LLFIND                             88 -     88 -      88 -     88 -
 proc LLPRTSTRING                         0 -      0 -       0 -      0 -
 proc LLPRTTOKEN                          0 -      0 -       0 -      0 -
 proc LLSKIPTOKEN                         0 -                          0 -
 proc LLSKIPNODE                          0 -                          0 -
 proc LLSKIPBOTH                          0 -                          0 -
 proc LLFATAL                             0 -                          0 -
 proc GET_CHARACTER                       0 -      0 -       0 -      0 -
 func MAKE_TOKEN                          0 -      0 -       0 -      0 -
 proc LLNEXTTOKEN                       100      100       100      100
 proc LLMAIN                             62 -     47 -      47 -     56 -
 body LL_COMPILE                        100                          100
-----------------------------------------------------------------------------
proc LL_COMPILE                          49 -     44 -      45 -     48 -
-----------------------------------------------------------------------------
Source file => [-.adalex2]ll_tokens.ada                    Instr => (A,N,N,N)

pack LL_TOKENS
 proc ADVANCE                            83 -     71 -      74 -     79 -
-----------------------------------------------------------------------------
pack LL_TOKENS                           83 -     71 -      74 -     79 -
=============================================================================
                    ...
=============================================================================
OVERALL SUMMARY :                        46 -     44 -      44 -     46 -
=============================================================================
Number of partially instrumented or dropped compilation units :  0

This summary was generated at 1991-08-14 13:34:48, based on the TCMON
execution profile listing file sample_out.dat.
The profile listing was produced at 1991-08-14 13:32:48, and the actual
TCPRE instrumentation was performed at 1991-08-14 13:15:27.

There were 103 places out of 128 where the coverage percentage was
below the selected warning level 100 %.
```

Figure 24-6. TCMON Coverage Summary

25. TCAT/Ada, TCAT-PATH, S-TCAT/Ada, TSCOPE, & TDGen

These tools are part of the Software TestWorks toolset that also includes SMARTS, CAPBAK, and EXDIFF for regression testing. TCAT/Ada, TCAT-PATH, and S-TCAT/Ada provide structural coverage analysis at unit and integration levels. TSCOPE provides a graphical animation of the coverage achieved and software data visualization relating to software quality, software performance, and software capacity. TDGen is a test data generation tool capable of generating test data randomly, sequentially, or using specified values based on a user-defined template.

TRACKER and STATIC are two additional tools due for release in fall 1992. TRACKER supports problem reporting. STATIC, which currently only operates on C code, performs static analysis to look for such items as missing break statements, initialized or unaccessed arrays and structures, loss of precision, and nonconformance with the ANSI C programming language standard.

25.1 Tool Overview

Software TestWorks has been marketed by Software Research, Inc.for over five years. Software Research also offers a range of software testing services, technical seminars, and programming language validation suites. The tools are available on a large number of operating platforms ranging from PCs to mainframes under Unix, MS-DOS, OS-2, and VMS operating systems. TCAT/Ada, TCAT-PATH, S-TCAT/Ada, and TDGen can each be invoked via a command line or through a windows-based graphical user interface in the OSF/Motif environment. TSCOPE requires the graphical user interface. Prices depend on the operating environment and, at the time of examination, started at $4,900 for TCAT, TCAT-PATH, S-TCAT, and TSCOPE together. Over 2,500 licenses have been sold for this group of tools. Prices for TDGen started at $500. Tool users are supported by both a newsletter and hot-line support.

The examinations were performed on a Sun-4 copy of TCAT/Ada Version 7.3, S-TCAT/Ada Version 7.6, TCAT-PATH Release 7, and TSCOPE Release 2. These are all recently released versions and still subject to beta testing. The final tool examined was TDGen Release 3.2.

25.1.1 TCAT/Ada and S-TCAT/Ada Overview

TCAT/Ada provides for segment or branch coverage analysis at the unit level and STCAT/Ada for call-pair coverage analysis at the module integration level. Both of these tools, however, operate similarly. The user starts by instrumenting the code under test. Some control over the extent of instrumentation to be performed is provided by allowing the user to specify a list of modules that are to be excluded from instrumentation. STCAT/Ada additionally allows the user to provide a file containing a list of function calls to be excluded and a switch that allows the user to specify the number of characters in a function name that shall be treated as distinct. In addition to the instrumented code, the instrumentor yields a reference listing that shows segment (or function) markers and instrumentation statistics on a module-by-module basis.

The instrumented program is then compiled and linked with a provided run-time file. Software Research provides different run-time routines to allow some flexibility in the behavior and performance of the instrumented program. For example, standard trace file processing is performed without internal processing or buffering, and the trace file is the full, unedited trace of program execution. One option provides for in-place data reduction with the entire coverage statistics being accumulated in memory and the trace file written after the program exits. Another option allows the user to turn trace sampling on and off after a specified number of trace records have been registered in memory. A special multi-tasking run-time routine is needed to handle instrumented processes that run in parallel; in this case, a trace file is produced for parent and child processes.

When running, the instrumented program queries the user for the name of the trace file to which execution data will be written. This trace file is subsequently used by a reporting utility to list the overall coverage achieved, identify hit and not-hit segments, and produce histograms showing the frequency distribution of segment or function hits, using either linear or logarithmic scales. All information is given in terms of the segment (or function) numbers shown in the reference listing. In the case of instrumented processes that run in parallel, a special utility is provided for preprocessing of the generated trace file(s). This utility splits tasking and non-tasking trace records into several files so that a trace file for each task is created.

The reporting function can handle several trace files at the same time and provides for cumulative coverage analysis by archiving trace file information into an archive file. With the exception of information on the sequence in which segments were hit, archive files con-

tain the same data as a trace file. For cumulative reporting, information on newly hit or newly missed items is provided. Finally, the user can restrict coverage reporting to a specified set of modules, request reporting on past coverage using only named archive files, and cause named modules be deleted from the archive. The results of analysis can be used to control the extent of subsequent instrumentation. This is achieved via a threshold switch that causes any module with percentage coverage greater than or equal to this threshold to be written to the de-instrumented file. (The user can specify the threshold value, or use the default value of 85%.)

TCAT/Ada and S-TCAT/Ada both provide a utility for creating null archive files. This is used to ensure that the coverage reporting covers all modules in a program whether or not they have been executed. This prevents artificially high initial levels of coverage.

TCAT/Ada and S-TCAT/Ada also include an additional utility that takes the output of the instrumentor to generate, respectively, directed graphs and call graphs of the code under test. Both of these graphs are presented in textual list form unless the graphical user interface is used. When the graphical user interface is used, a graphical representation of the appropriate diagram can be supported by displays that show, for example, the associated code, path statistics, and standards limits. In the case of S-TCAT/Ada, the call graph can be started from a user-specified root node and the outputs of instrumentation of several source files can be combined to generate a call-graph for the whole program.

25.1.2 TCAT-PATH Overview

TCAT-PATH differs from TCAT/Ada and S-TCAT/Ada in that coverage reporting addresses the paths executed and is only provided on a single trace file; archive files are not supported. As with the two previous tools, the user can limit the amount of instrumentation performed by providing a file containing the names of functions not to instrument. TCAT-PATH, however, allows the user to further limit instrumentation by inserting flags in the code that switch instrumentation on and off; these flags are given as a special type of comment and can be left permanently in the code. The user also can specify that instrumentation of empty edges be suppressed.

In addition to the instrumented code and reference listing, the instrumentor generates a separate file of directed graph information for each module. The same utilities as are provided with TCAT/Ada are available for using this information to draw directed graphs. The information is also used to support the following utilities:

- apg. This automatic path generator gives the complete set of paths for a named module. The user can request that only *basis paths* are listed; that is, those paths that have no iteration. Additional options include presentation of a set of path statistics, specification of the maximum limit on the number of paths to generate, and specification of pairs of segments not to include in paths. In complex cases, apg can be applied to a subgraph instead of complete directed graphs.
- pathcon. Presents the logical conditions, and associated predicates, that cause a path in a particular module to be executed. It can be invoked for a single path, a set or range of paths, or all paths.
- pathcover. Presents the *essential paths* in a module, that is, those required to be executed to achieve 100% coverage. Essential paths are determined based on the order of segment occurrence where this order may be adjusted by sorting path information on various criteria. Population statistics on each segment are available.
- cyclo. Computes the cyclomatic complexity for the named module.

Since these utilities are invoked for a single module, they can require many repetitive operations on the part of the user. Consequently, two additional utilities, DoPTH and DoCYC, allow the user to request that, respectively, apg and cyclo are applied to all appropriate modules. DoPIC provides a similar facility for drawing directed graphs

The instrumented program is compiled and linked as before and, again, a number of special run-time routines are available to provide some control over its behavior and performance. When executed, the instrumented program generates a trace file for subsequent coverage analysis. The basic coverage utility analyzes this trace file to report on the path coverage achieved for a named module. For each path in the module, the coverage report specifies whether it was executed and, if so, how many times, together with an overall coverage value. A special DoRPT utility invokes the coverage analyzer for all modules supported by apg-generated path information.

25.1.3 TSCOPE Overview

TSCOPE is used with the trace files produced by TCAT/Ada, TCAT-PATH, or S-TCAT/Ada to animate test coverage. All TSCOPE commands are treated as primitives that are invoked to present a variety of displays. Consequently, all instrumented modules can be reported on a single X-Window screen and different kinds of reporting can be selected for different modules. (Each program module can be instrumented for either segment, path, or call-pair coverage; since these are incompatible, only one type of information can be reported from each module.) Displays are positioned on the screen using a special configuration file or interactively using TSCOPE menu options.

The following types of dynamic displays are available:

- TSc1dig. Provides dynamic display of segment or path coverage data on the directed graph for a named module. Five different animation styles are available.
- TShisto. Provides a dynamic linear histogram of segment (or path) coverage and call-pair coverage for a named module. This histogram reflects the percentage of times a segment is hit. It is supported by data on the number of times a segment is hit and the current segment coverage.
- TSlhisto. Provides a dynamic logarithmic histogram of segment (or path) coverage and call-pair coverage for a named module. This display provides similar information to the linear histogram display, except that it shows the differences between relative segment hit counts more clearly.
- TSs0cg and TSs1cg. Provides dynamic display of coverage data for a named module on one of two types of call tree. TSs0cg presents a call tree that shows each distinct link between an invoking and invoked module; it is used for showing coverage with respect to the percentage of modules invoked. TSs1cg gives only one line for each invoking-invoked relationship, regardless of the number of connections, and is used for display of call-pair coverage data.
- TSstrip. Provides a dynamic strip chart that shows the accumulation of segment (or path) coverage for a named module during a single test. This chart is supported by data on the percentage of segments (or paths) that have been hit and the number of times the module is called.

TSCOPE also supports two static displays. These are provided by the utility available with TCAT/Ada and TCAT-PATH for graphical display of directed graphs, and that available with S-TCAT/Ada for graphical display of call trees. A number of additional utilities support display management.

25.1.4 TDGen Overview

TDGen generates test data, or test files, from user defined specifications. It is particularly useful for generating the large amounts of test data needed in stress testing.

TDGen works with two files. The Template File tells TDGen how to generate test data based on data supplied in a Values File. TDGen replaces variable fields, called descriptors, in the template with values from the Values File. Descriptors may be user defined or take one of the predefined values (ASCII, alpha, decimal, and real). In the Values File, descriptors are associated with potential values. Special notations are provided for specifying ranges of values and handling comments, blanks, and other white space. Once the Values and Template files have been created, the user can invoke test data generation in one of three ways to specify how values should be taken for the descriptors in the Template File:

- Specifically. The values for all or some of the descriptors are specified by integers.
- Sequentially. Values are taken sequentially from the Values File to generate every possible combination of the given values.
- Randomly. Selects values from the Values File randomly by taking one value from each field in the file at random. For each field name encountered in the Template File, a uniformly distributed random number is used to select a particular value from those corresponding.

Additionally, the user can request TDGen to tabulate the number of possible test data combinations that will be generated to allow a review of the size of the results before generation commences.

25.2 Observations

Ease of use. A user can interact with these tools using either a command-line interface or a series of menus. Context-sensitive help and help frames discussing each function are provided. No special knowledge is required to use these tools.

TCAT/Ada, S-TCAT/Ada, and TCAT-PATH all provide a number of utilities that can be invoked for individual program units. In most cases, a special utility is available to apply the utility for all available program units using a single command. Additionally, TCAT-PATH supports Unix-like *make* files to facilitate repetitive compilation and linking tasks.

A limited amount of tailoring is provided by the use of configuration files. These allow the user to adjust, for example, setting the maximum number of nodes to process, formatting options for diagraph display, and default path names.

Reports are generally well-structured. Since segments, paths, and call-pairs are referred to by number, however, a user must refer back to the various reference listings to identify the subject of each reference.

Documentation and user support. The tools are supported by extensive documentation that includes guidelines on appropriate minimum coverage levels.

Instrumentation overhead. TCAT/Ada, TCAT-PATH, and S-TCAT/Ada instrument the contents of files specified as part of the tool invocation. In each case, all code is instrumented the same way. For TCAT/Ada, the vendor recommends a capacity of up to 2,500 segments (approximately 20,000 lines of code). The vendor estimates the size or performance overhead for instrumentation at 20% to 30%, although this can be higher for very complex programs. For the Ada Lexical Analyzer Generator, TCAT/Ada, TCAT-PATH,

and S-TCAT/Ada instrumentation gave, respectively, 37%, 37%, and 28% increases in source code size with corresponding increases of 15%, 12%, and 15% for executable code. Versions of TCAT and S-TCAT that accomplish various levels of in-place buffering to enhance performance are available for C programs. Similar support is not available for the Ada versions.

Ada restrictions. The TCAT/Ada and S-TCAT/Ada instrumentors have been validated against the Ada validation suite, a set of programs that test compliance with the Ada standard.

TCAT/Ada and S-TCAT/Ada do not support conditional expressions in Ada; such expressions must be expanded to the explicit *if-then-else* form. TCAT-PATH does not handle multiple instances of a task or exception handling. Variant records, compound conditions, and the terminate alternative of a selective wait are not instrumented by any of these tools.

Problems encountered. Problems encountered with the runtime files and installation instructions of earlier releases of these tools have been fixed. For each of TCAT/Ada, S-TCAT/Ada, and TCAT-PATH, incorrectly inserted instrumentation statements prevented compilation. In most cases, however, these errors were relatively easy to correct manually. The TCAT-PATH pathcon utility gave a segmentation fault after processing the first record of generated trace files, and the coverage reporting utility could not report the coverage achieved for some program units; TSstrip was the only utility that consistently worked as expected, though TSc1dig worked as long as certain command options were not given. It was not possible to get several windows displayed on a screen. Software Research are investigating these problems. In TDGen, errors in values and template file, or in the specification of program options, caused the program to hang.

25.3 Recent Changes

Software TestWorks has been integrated with IBM's AIX Software Development Environment Workbench/6000.

25.4 Sample Outputs

Figures 25-1 through 25-26 provide sample outputs from these tools.

```
------------------------------------------------------------------------
-- TCAT/Ada, Release 2.1 for SUN (09/16/92).
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- SEGMENT REFERENCE LISTING Fri Sep 25 13:26:32 1992
                                        ...
      procedure  LLFATAL  is
         --  To recover from syntactic error, terminate compilation
      begin
--* Module new_ll_compile.LLFATAL *--
--* Segment 1 <> *--
         PUT( STANDARD_ERROR, "*** Fatal " );
         LLPRTTOKEN;
         PUT( STANDARD_ERROR, " found in line ");
         PUT( STANDARD_ERROR, LLCURTOK.LINENUMBER, 1 );
         PUT_LINE( STANDARD_ERROR, " -- terminating translation." );
         raise  PARSING_ERROR;
      end  LLFATAL;
--* End module new_ll_compile.LLFATAL *--
                                        ...
      begin   -- TESTSYNCH
--* Module new_ll_compile.TESTSYNCH *--
--* Segment 1 <> *--
         while  LLSTACK(LLSENTPTR).DATA.SYNCHINDEX = 0  loop
--* Segment 2 <> *--
            -- no synch info there
            if  LLSTACK(LLSENTPTR).PARENT /= 0  then
--* Segment 3 <> *--
               -- there really is a parent
               LLSENTPTR := LLSTACK(LLSENTPTR).PARENT;
            else
--* Segment 4 <> *--
               LLFATAL;
            end if;
         end loop;
--* Segment 5 <> *--
         SYNCHRONIZE;
      end  TESTSYNCH;
--* End module new_ll_compile.TESTSYNCH *--
                                        ...
      begin  -- LL_COMPILE
--* Module new_ll_compile.LL_COMPILE *--
--* Segment 1 <> *--
         OPEN( LLSOURCE, IN_FILE, "SOURCE" );
         LLMAIN;
         CLOSE( LLSOURCE );
      end  LL_COMPILE;
--
-- END OF TCAT/Ada REFERENCE LISTING
--
```

Figure 25-1. TCAT/Ada Reference Listing for LL_COMPILE

```
--------------------------------------------------------------------
-- TCAT/Ada, Release 2.1 for SUN (09/16/92).
--
-- INSTRUMENTATION STATISTICS
--
-- Module                  # segments     # statements  # Conditional statements
--
-- new_ll_compile.LLFIN           8             11              3
-- new_ll_compile.LLPRT           5              5              2
-- new_ll_compile.LLPRT           3              4              1
-- new_ll_compile.LLSKI           1              7              0
-- new_ll_compile.LLSKI           1              8              0
-- new_ll_compile.LLSKI           1              9              0
-- new_ll_compile.LLFAT           1              6              0
-- new_ll_compile.GET_C           4              7              1
-- new_ll_compile.CVT_S           5              5              2
-- new_ll_compile.MAKE_          15             18              4
-- new_ll_compile.LLNEX           3              5              1
-- new_ll_compile.BUILD          15             32              5
-- new_ll_compile.BUILD           3              5              1
-- new_ll_compile.READG          11             24              5
-- new_ll_compile.ERASE           5              7              2
-- new_ll_compile.MATCH           7              6              3
-- new_ll_compile.EXPAN          13             21              6
-- new_ll_compile.SYNCH          19             29              9
-- new_ll_compile.TESTS           5              5              2
-- new_ll_compile.PARSE          18             29              6
-- new_ll_compile.LLMAI           1              2              0
-- new_ll_compile.LL_CO           1              3              0
```

Figure 25-2. TCAT/Ada Instrumentation Statistics for LL_COMPILE

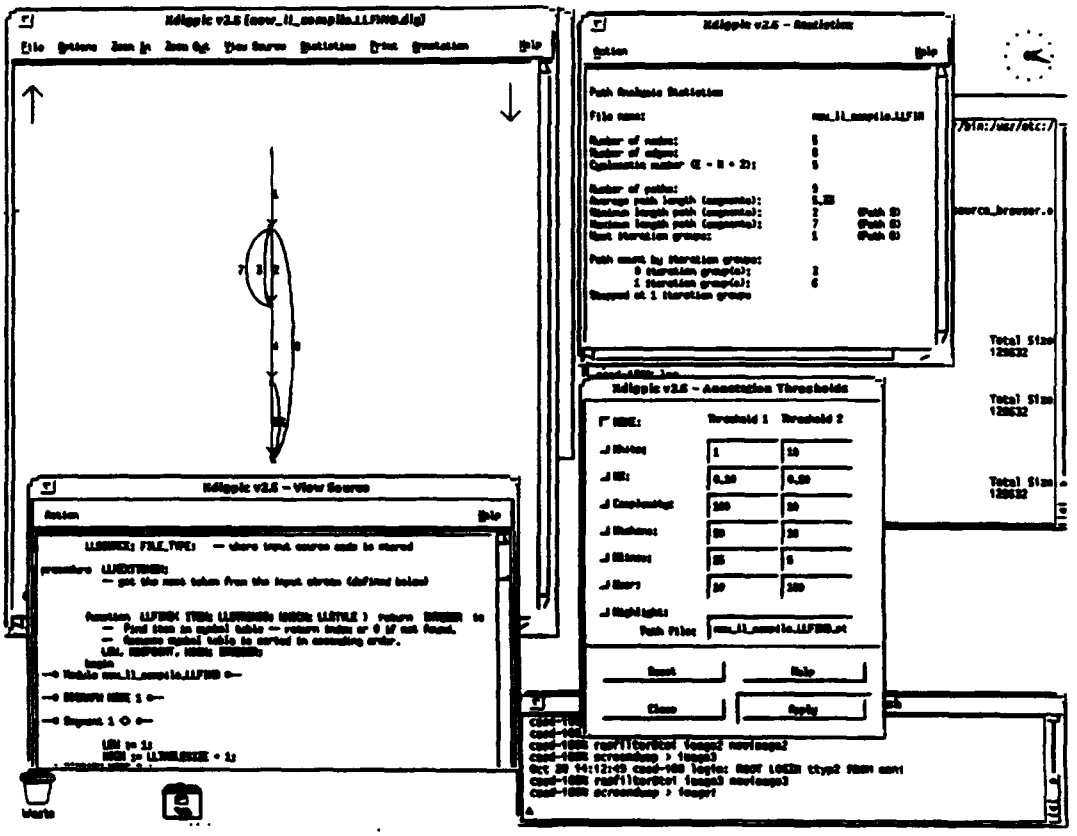


Figure 25-3. TCAT/Ada Directed Graph for LLFIND from LL_COMPILE

```
Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

Selected COVER System Option Settings:
[-c]  Cumulative Report     -- NO
[-p]  Past History Report   -- NO
[-n]  Not Hit Report        -- YES
[-H]  Hit Report            -- YES
[-nh] Newly Hit Report      -- NO
[-nm] Newly Missed Report   -- NO
[-h]  Histogram Report      -- YES
[-l]  Log Scale Histogram   -- YES
[-z]  Reference Listing C1  -- NO
Options read:   4

TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

C1 Segment Hit Report.

No.   Module Name:            Segment Coverage Status:
1     new_ll_compile.LL_COMPILE       All Segments Hit.  C1 = 100%
2     new_ll_compile.LLMAIN           All Segments Hit.  C1 = 100%
3     new_ll_compile.READGRAM         All Segments Hit.  C1 = 100%
4     new_ll_compile.BUILDRIGHT       1   2   3   4   5   6   7   8   10
                                      11  12  13  15
5     new_ll_compile.BUILDSELECT      All Segments Hit.  C1 = 100%
6     new_ll_compile.PARSE            1   2   3   4   10  11  12  14  15
                                      16  18
7     new_ll_compile.LLFIND           1   2   3   4   5   7   8
8     new_ll_compile.LLNEXTTOKEN      All Segments Hit.  C1 = 100%
9     new_ll_tokens.ADVANCE           1   2   3   4   5   6   7   8
10    new_ll_tokens.SCAN_PATTERN      1   2   3   4   5   6   7   8   9
                                      37  38  39  40  43  45  47  49  61
                                      62  63  64  65  66  67  68  69  70
                                      71  74  75  76  84  88  89  90  91
                                      98  99  104 105 106
11    new_ll_compile.GET_CHARACTER    All Segments Hit.  C1 = 100%
                                    ...
42    ll_sup_body.EMIT_CHAR           1   10
43    ll_sup_body.EMIT_PATTERN_MATCH  1   9   10  12  13  17  18  20  22
                                      24  25  27  28  29
44    ll_sup_body.EMIT_CONCAT_RIGHT   1   5
45    ll_sup_body.EMIT_CONCAT_CASES   1   2   3   5   7

Number of Segments Hit:         308
Total Number of Segments:       529
C1 Coverage Value:             58.22%
```

Figure 25-4. TCAT/Ada Segment Coverage Report using test1.lex

```
TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

C1 Segment Not Hit Report.

No.   Module Name:              Segment Coverage Status:
1     new_ll_compile.LL_COMPILE       All Segments Hit.  C1 = 100%
2     new_ll_compile.LLMAIN           All Segments Hit.  C1 = 100%
3     new_ll_compile.READGRAM         All Segments Hit.  C1 = 100%
4     new_ll_compile.BUILDRIGHT       9   14
5     new_ll_compile.BUILDSELECT      All Segments Hit.  C1 = 100%
6     new_ll_compile.PARSE            5    6    7    8    9   13   17
7     new_ll_compile.LLFIND           6
8     new_ll_compile.LLNEXTTOKEN      All Segments Hit.  C1 = 100%
9     new_ll_tokens.ADVANCE           9   10   11
10    new_ll_tokens.SCAN_PATTERN      10   11   12   13   14   15   16   17   18
                                      19   20   21   22   23   24   25   26   27
                                      28   29   30   31   32   33   34   35   36
                                      41   42   44   46   48   50   51   52   53
                                      54   55   56   57   58   59   60   72   73
                                      77   78   79   80   81   82   83   85   86
                                      87   92   93   94   95   96   97  100  101
                                      102  103  107  108  109
11    new_ll_compile.GET_CHARACTER    All Segments Hit.  C1 = 100%
                                    ...
40    ll_sup_body.EMIT_SCAN_PROC      2
41    ll_sup_body.EMIT_SCAN_SELECT    15  17
42    ll_sup_body.EMIT_CHAR           2    3    4    5    6    7    8    9   11
                                      12
43    ll_sup_body.EMIT_PATTERN_MATCH  2    3    4    5    6    7    8   11   14
                                      15   16   19   21   23   26   30   31   32
44    ll_sup_body.EMIT_CONCAT_RIGHT   2    3    4
45    ll_sup_body.EMIT_CONCAT_CASES   4    6    8

Number of Segments Not Hit:      221
Total Number of Segments:          529
C1 Coverage Value:              58.22%
```

Figure 25-4 continued: TCAT/Ada Segment Coverage Report using test1.lex

```
TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
Segment Level Histogram for Module: new_ll_compile.LL_COMPILE

                        +-------------------------------------------------+
                        | Number of Executions, Normalized to Maximum
                        | (Maximum =      1 Hits)   X = One Hit
                        | (Scale:  100.000       Each X =    0.020 Hits)
 Segment  Number Of     |
 Number   Executions   >-1-------20--------40--------60--------80-------100
+-----------------------+-------------------------------------------------+
                        |
    1             1     | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                        |
+-----------------------+-------------------------------------------------+
Average Hits per Executed Segment:    1.0000
C1 Value for this Module:           100.0000
                                ...
TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
Segment Level Histogram for Module: new_ll_compile.BUILDRIGHT

                        +-------------------------------------------------+
                        | Number of Executions, Normalized to Maximum
                        | (Maximum =    174 Hits)   X = One Hit
                        | (Scale:    0.575       Each X =    3.480 Hits)
 Segment  Number Of     |
 Number   Executions   >-1-------20--------40--------60--------80-------100
+-----------------------+-------------------------------------------------+
                        |
    1            64     | XXXXXXXXXXXXXXXXX
    2           174     | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    3           174     | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    4            50     | XXXXXXXXXXXXX
    5            61     | XXXXXXXXXXXXXXXX
    6            46     | XXXXXXXXXXXX
    7            13     | XXX
    8             4     | X
    9   *               |
   10            79     | XXXXXXXXXXXXXXXXXXXX
   11            95     | XXXXXXXXXXXXXXXXXXXXXXXX
   12           144     | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   13            30     | XXXXXXXX
   14   *               |
   15            64     | XXXXXXXXXXXXXXXXX
                        |
+-----------------------+-------------------------------------------------+
        (* = Zero Hits)
Average Hits per Executed Segment:   76.7692
C1 Value for this Module:            86.6667
                                ...
```

Figure 25-4 continued: TCAT/Ada Segment Coverage Report using test1.lex

```
Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

Selected COVER System Option Settings:
[-c]  Cumulative Report     -- YES
[-p]  Past History Report   -- NO
[-n]  Not Hit Report        -- YES
[-H]  Hit Report            -- YES
[-nh] Newly Hit Report      -- YES
[-nm] Newly Missed Report   -- YES
[-h]  Histogram Report      -- NO
[-l]  Log Scale Histogram   -- YES
[-z]  Reference Listing C1  -- NO
Options read:  6

TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
```

		Current Test			Cumulative Summary		
Module Name:	Number Of Segments:	No. Of Invokes	No. Of Segments Hit	C1% Cover	No. Of Invokes	No. Of Segments Hit	C1% Cover
new_ll_compile.LL_CON	1	1	1	100.00	2	1	100.00
new_ll_compile.LLMAIN	1	1	1	100.00	2	1	100.00
new_ll_compile.READGR	11	1	11	100.00	2	11	100.00
new_ll_compile.BUILDR	15	64	13	86.67	128	13	86.67
new_ll_compile.BUILDS	3	64	3	100.00	128	3	100.00
new_ll_compile.PARSE	18	1	11	61.11	2	11	61.11
new_ll_compile.LLFIND	8	312	8	100.00	510	8	100.00
new_ll_compile.LLNEXT	3	221	3	100.00	355	3	100.00
new_ll_tokens.ADVANCE	11	221	9	81.82	355	9	81.82
new_ll_tokens.SCAN_PA	109	455	46	42.20	712	52	47.71
new_ll_compile.GET_CH	4	1385	4	100.00	2250	4	100.00
new_ll_tokens.CHAR_AD	5	1238	3	60.00	2018	3	60.00
new_ll_tokens.CURRENT	1	220	1	100.00	353	1	100.00
new_ll_compile.MAKE_T	15	220	13	86.67	353	13	86.67
new_ll_compile.CVT_ST	5	220	5	100.00	353	5	100.00
new_ll_compile.EXPAND	13	461	11	84.62	715	11	84.62
new_ll_compile.MATCH	7	461	5	71.43	715	5	71.43
new_ll_compile.ERASE	5	707	5	100.00	1105	5	100.00
new_ll_tokens.LOOK_AH	5	84	3	60.00	123	3	60.00
ll_actions.LLTAKEACTI	69	429	36	52.17	659	36	52.17
		...					
ll_sup_body.TAIL	18	2	4	22.22	2	4	22.22
ll_sup_body.EMIT_ALT_	7	12	7	100.00	12	7	100.00
Totals	614	7569	384	62.54	11884	391	63.68

Figure 25-5. TCAT/Ada Segment Coverage Report using test1.lex & sample.lex

```
(c) Copyright 1990 by Software Research, Inc.

C1 Segment Hit Report.

No.   Module Name:              Segment Coverage Status:
1     new_ll_compile.LL_COMPILE       All Segments Hit.  C1 = 100%
2     new_ll_compile.LLMAIN           All Segments Hit.  C1 = 100%
3     new_ll_compile.READGRAM         All Segments Hit.  C1 = 100%
4     new_ll_compile.BUILDRIGHT       1   2   3   4   5   6   7   8   10
                                      11  12  13  15
5     new_ll_compile.BUILDSELECT      All Segments Hit.  C1 = 100%
6     new_ll_compile.PARSE            1   2   3   4   10  11  12  14  15
                                      16  18
7     new_ll_compile.LLFIND           All Segments Hit.  C1 = 100%
8     new_ll_compile.LLNEXTTOKEN      All Segments Hit.  C1 = 100%
9     new_ll_tokens.ADVANCE           1   2   3   4   5   6   7   8   9
10    new_ll_tokens.SCAN_PATTERN      1   2   3   4   5   6   7   8   9
                                      10  11  12  13  14  15  16  17  28
                                      32  37  38  39  40  43  45  47  49
                                      61  62  63  64  65  66  67  68  69
                                      70  71  74  75  76  84  88  89  90
                                      91  98  99  104 105 106 107
11    new_ll_compile.GET_CHARACTER    All Segments Hit.  C1 = 100%
12    new_ll_tokens.CHAR_ADVANCE      1   3   5
13    new_ll_tokens.CURRENT_SYMBOL    All Segments Hit.  C1 = 100%
                                    ...
50    ll_sup_body.TAIL                1   5   14  16
51    ll_sup_body.EMIT_ALT_CASES      All Segments Hit.  C1 = 100%

Number of Segments Hit:          391
Total Number of Segments:        614

C1 Coverage Value:             63.68%


TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

C1 Segment Newly Hit Report.

No.   Module Name:              Segment Coverage Status:
7     new_ll_compile.LLFIND           6
9     new_ll_tokens.ADVANCE           9
10    new_ll_tokens.SCAN_PATTERN      10  11  12  13  14  15  16  17  28
                                      32  107
                                    ...
50    ll_sup_body.TAIL                1   5   14  16
51    ll_sup_body.EMIT_ALT_CASES      1   2   3   4   5   6   7
```

Figure 25-5 continued: TCAT/Ada Segment Coverage Report using test1.lex & sample.lex

```
TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

C1 Segment Not Hit Report.

No.   Module Name:              Segment Coverage Status:
1     new_ll_compile.LL_COMPILE       All Segments Hit.  C1 = 100%
2     new_ll_compile.LLMAIN           All Segments Hit.  C1 = 100%
3     new_ll_compile.READGRAM         All Segments Hit.  C1 = 100%
4     new_ll_compile.BUILDRIGHT       9   14
5     new_ll_compile.BUILDSELECT      All Segments Hit.  C1 = 100%
6     new_ll_compile.PARSE            5    6    7    8    9   13   17
7     new_ll_compile.LLFIND           All Segments Hit.  C1 = 100%
8     new_ll_compile.LLNEXTTOKEN      All Segments Hit.  C1 = 100%
9     new_ll_tokens.ADVANCE           10   11
10    new_ll_tokens.SCAN_PATTERN      18   19   20   21   22   23   24   25   26
                                      27   29   30   31   33   34   35   36   41
                                      42   44   46   48   50   51   52   53   54
                                      55   56   57   58   59   60   72   73   77
                                      78   79   80   81   82   83   85   86   87
                                      92   93   94   95   96   97  100  101  102
                                      103  108  109
11    new_ll_compile.GET_CHARACTER    All Segments Hit.  C1 = 100%
12    new_ll_tokens.CHAR_ADVANCE      2    4
13    new_ll_tokens.CURRENT_SYMBOL    All Segments Hit.  C1 = 100%
14    new_ll_compile.MAKE_TOKEN       10   15
                                    ...
48    ll_sup_body.RESOLVE_AMBIGUITY   4    5    8    9   10   11   12   13   14
                                      15   16   17   18   19
49    ll_sup_body.RESTRICT            4    7   13   14   15   16   22
50    ll_sup_body.TAIL                2    3    4    6    7    8    9   10   11
                                      12   13   15   17   18
51    ll_sup_body.EMIT_ALT_CASES      All Segments Hit.  C1 = 100%

Number of Segments Not Hit:      223
Total Number of Segments:        614
C1 Coverage Value:             63.68%


TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

C1 Segment Newly Missed Report.

No.     Module Name:          Segment Coverage Status:
10      new_ll_tokens.SCAN_PATTERN    66   67   68   69   70   71
40      ll_sup_body.EMIT_SCAN_PROC    6
```

Figure 25-5 continued: TCAT/Ada Segment Coverage Report using test1.lex & sample.lex

```
TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
Segment Level Histogram for Module: new_ll_compile.LL_COMPILE

                        +-------------------------------------------------------+
                        | Logarithm of Executions, Normalized to Maximum
                        | (Maximum =       2 Hits)
 Segment  Number Of     |
 Number   Executions    >-----------1----------10-----20----30---40--80-100
+-----------------------+-------------------------------------------------------+
                        |
    1              2    | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                        |
+-----------------------+-------------------------------------------------------+

Average Hits per Executed Segment:    2.0000
C1 Value for this Module:           100.0000
                                    ...

TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
Segment Level Histogram for Module: new_ll_compile.READGRAM

                        +-------------------------------------------------------+
                        | Logarithm of Executions, Normalized to Maximum
                        | (Maximum =    1280 Hits)
 Segment  Number Of     |
 Number   Executions    >-----------1----------10-----20----30---40--80-100
+-----------------------+-------------------------------------------------------+
                        |
    1              2    | XXXXXXX
    2             64    | XXXXXXXXXXXXXXXXXXXX
    3           1280    | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    4             64    | XXXXXXXXXXXXXXXXXXXX
    5             12    | XXXXXXXXXXXX
    6             52    | XXXXXXXXXXXXXXXXXXX
    7              2    | XXXXXXX
    8            128    | XXXXXXXXXXXXXXXXXXXXXXXXX
    9              2    | XXXXXXX
   10             52    | XXXXXXXXXXXXXXXXXXX
   11              2    | XXXXXXX
                        |
+-----------------------+-------------------------------------------------------+

Average Hits per Executed Segment:  150.9091
C1 Value for this Module:           100.0000
                                    ...
```

Figure 25-5 continued: TCAT/Ada Segment Coverage Report using test1.lex & sample.lex

```
------------------------------------------------------------------------
-- TCAT-PATH/Ada, Release 2.1 for SUN (09/18/92).
--
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
--
-- SEGMENT REFERENCE LISTING Mon Oct 26 13:09:12 1992
                                        ...
procedure  LLNEXTTOKEN;
         -- get the next token from the input stream (defined below)

      function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
         --  Find item in symbol table -- return index or 0 if not found.
         --  Assumes symbol table is sorted in ascending order.
         LOW, MIDPOINT, HIGH: INTEGER;
      begin
--* Module new_ll_compile.LLFIND *--
--* DIGRAPH NODE 1 *--
--* Segment 1 <> *--
         LOW := 1;
         HIGH := LLTABLESIZE + 1;
--* DIGRAPH NODE 2 *--
         while  LOW /= HIGH  loop
--* Segment 2 <> *--
            MIDPOINT := (HIGH + LOW) / 2;
--* DIGRAPH NODE 3 *--
            if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
--* Segment 3 <> *--
               HIGH := MIDPOINT;
            elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
--* Segment 4 <> *--
--* DIGRAPH NODE 4 *--
               if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
--* Segment 5 <> *--
                  return( MIDPOINT );
               else
--* Segment 6 <> *--
                  return( 0 );
               end if;
            else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
--* Segment 7 <> *--
               LOW := MIDPOINT + 1;
            end if;
         end loop;
--* Segment 8 <> *--
         return( 0 );  -- item is not in table
      end  LLFIND;
--* DIGRAPH NODE 5 *--
--* End module new_ll_compile.LLFIND *--
                                        ...
--
-- END OF TCAT-PATH/Ada REFERENCE LISTING
```

Figure 25-6. TCAT-PATH Segment and Node Reference Listing for LL_COMPILE

```
-------------------------------------------------------------------------
-- TCAT-PATH/Ada, Release 2.1 for SUN (09/18/92).
--
-- INSTRUMENTATION STATISTICS
--
-- Module                  # segments     # statements  # Conditional statements
--
-- new_ll_compile.LLFIN          8             11              3
-- new_ll_compile.LLPRT          5              5              2
-- new_ll_compile.LLPRT          3              4              1
-- new_ll_compile.LLSKI          1              7              0
-- new_ll_compile.LLSKI          1              8              0
-- new_ll_compile.LLSKI          1              9              0
-- new_ll_compile.LLFAT          1              6              0
-- new_ll_compile.GET_C          4              7              1
-- new_ll_compile.CVT_S          5              5              2
-- new_ll_compile.MAKE_         15             18              4
-- new_ll_compile.LLNEX          3              5              1
-- new_ll_compile.BUILD         15             32              5
-- new_ll_compile.BUILD          3              5              1
-- new_ll_compile.READG         11             24              5
-- new_ll_compile.ERASE          5              7              2
-- new_ll_compile.MATCH          7              6              3
-- new_ll_compile.EXPAN         13             21              6
-- new_ll_compile.SYNCH         19             29              9
-- new_ll_compile.TESTS          5              5              2
-- new_ll_compile.PARSE         18             29              6
-- new_ll_compile.LLMAI          1              2              0
-- new_ll_compile.LL_CO          1              3              0
```

Figure 25-7. TCAT-PATH Instrumentation Statistics for LLFIND

```
cyclo  [Release 3.3 -- 9/26/90]

Cyclomatic Number  =  Edges - Nodes + 2  =  8 - 5 + 2  =  5
```

Figure 25-8. TCAT-PATH Cyclomatic Complexity of Function LLFIND

```
new_ll_compile.LLFIND   8
new_ll_compile.LLPRTSTRING     5
new_ll_compile.LLPRTTOKEN      3
new_ll_compile.LLSKIPTOKEN     1
new_ll_compile.LLSKIPNODE      1
new_ll_compile.LLSKIPBOTH      1
new_ll_compile.LLFATAL  1
new_ll_compile.GET_CHARACTER   4
new_ll_compile.CVT_STRING      5
new_ll_compile.MAKE_TOKEN      15
new_ll_compile.LLNEXTTOKEN     3
new_ll_compile.BUILDRIGHT      15
new_ll_compile.BUILDSELECT     3
new_ll_compile.READGRAM 11
new_ll_compile.ERASE    5
new_ll_compile.MATCH    7
new_ll_compile.EXPAND   13
new_ll_compile.SYNCHRONIZE     19
new_ll_compile.TESTSYNCH       5
new_ll_compile.PARSE    18
new_ll_compile.LLMAIN   1
new_ll_compile.LL_COMPILE      1
```

Figure 25-9. TCAT-PATH Segment Count for Each Module in LL_COMPILE

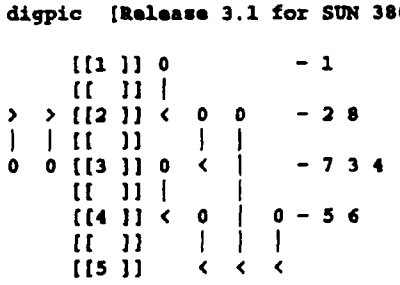


Figure 25-10. TCAT-PATH Digraph of Function LLFIND

```
apg [version 3.3 -- 09/02/92] -- paths for "new_ll_compile.LLFIND"

 1 2 4 5
 1 2 4 6
 1 2 3 <[ 2 3 7 ]> 8
 1 2 3 [[ 2 3 7 ]] 4 5
 1 2 3 [[ 2 3 7 ]] 4 6
 1 2 7 <[ 2 3 7 ]> 8
 1 2 7 [[ 2 3 7 ]] 4 5
 1 2 7 [[ 2 3 7 ]] 4 6
 1 8

------------------------------------------------------------
Total of 9 paths for 'new_ll_compile.LLFIND'.
```

Figure 25-11. TCAT-PATH All Paths for LLFIND

```
apg [version 3.3 -- 09/02/92] -- paths for "new_ll_compile.LLFIND"

 1 2 4 5
 1 2 4 6
 1 8

------------------------------------------------------------
Total of 3 paths for 'new_ll_compile.LLFIND'.
```

Figure 25-12. TCAT-PATH Basis Paths for LLFIND

```
apg [version 3.3 -- 09/02/92] -- paths for "new_ll_compile.LLFIND"

Path Analysis Statistics
File name:                      new_ll_compile.LLFIND.dig

Number of nodes:                5
Number of edges:                8
Cyclomatic number (E - N + 2):                  5

Number of paths:                9
Average path length (segments):                 5.33
Minimum length path (segments):                 2       (Path 9)
Maximum length path (segments):                 7       (Path 6)
Most iteration groups:                  1       (Path 8)

Path count by iteration groups:
      0 iteration group(s):             3
      1 iteration group(s):             6
Stopped at 1 iteration groups
```

Figure 25-13. TCAT-PATH Path Statistics for LLFIND

```
pathcover -- Path Coverage Utility. [Release 1.2 -- 9/91]
(c) Copyright 1991 by Software Research, Inc.

pathcover:  FIRST INSTANCE FOUND BY SEGMENT
Module:: "new_ll_compile.LLFIND"          Option:: "-f"
-------------------------------------------------------------------------------
#      Path# Path
-------------------------------------------------------------------------------
1      1     1 2 1
2      2     2 3 2
3      3     3 4 4
4      4     4 5 5
5      5     4 5 6
6      7     3 2 7
7      8     2 5 8
d

OR

pathcover:  POPULATION STATISTICS BY SEGMENT
Module:: "new_ll_compile.LLFIND"          Option:: "-c"
-------------------------------------------------------------------------------
Segment          # of paths
-------------------------------------------------------------------------------
  1          3
  2          2
  4          2
  5          1
  6          1
  8          1


pathcover:  FIRST INSTANCE FOUND BY SEGMENT
Module:: "new_ll_compile.LLFIND"          Option:: "-f"
-------------------------------------------------------------------------------
#      Path# Path
-------------------------------------------------------------------------------
1      1     1 2 4 5
2      2     1 2 4 6
3      3     1 8


pathcover:  LAST INSTANCE FOUND BY SEGMENT
Module:: "new_ll_compile.LLFIND"          Option:: "-l"
-------------------------------------------------------------------------------
#      Path# Path
-------------------------------------------------------------------------------
1      1     1 2 4 5
2      2     1 2 4 6
3      3     1 8
```

Figure 25-14. TCAT-PATH Path and Segment Information for LLFIND

```
Ct Test Coverage Analyser Version 2.1 (9/91)
      (c) Copyright 1991 by Software Research, Inc.
Module "new_ll_compile.BUILDRIGHT": 26 paths, 8 were hit in 64 invocations.
 30.77% Ct coverage

HIT/NOT-HIT REPORT
------------------
P#    Hits    Path text

1       25    1 2 3 4 10 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
2     None    1 2 3 4 10 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
3     None    1 2 3 4 11 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
4        1    1 2 3 4 11 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
5     None    1 2 3 5 10 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
6     None    1 2 3 5 10 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
7        5    1 2 3 5 11 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
8     None    1 2 3 5 11 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
9       11    1 2 3 6 10 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
10    None    1 2 3 6 10 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
11    None    1 2 3 6 11 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
12       4    1 2 3 6 11 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
13       8    1 2 3 7 10 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
14    None    1 2 3 7 10 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
15    None    1 2 3 7 11 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
16       1    1 2 3 7 11 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
17    None    1 2 3 8 10 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
18    None    1 2 3 8 10 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
19    None    1 2 3 8 11 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
20    None    1 2 3 8 11 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
21    None    1 2 3 9 10 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
22    None    1 2 3 9 10 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
23    None    1 2 3 9 11 12 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
24    None    1 2 3 9 11 13 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
25    None    1 2 14 <{ 2 3 4 10 12 13 11 5 6 7 8 9 14 }> 15
26       9    1 15
```

Figure 25-15. TCAT-PATH Coverage Report for BUILDRIGHT using test1.lex

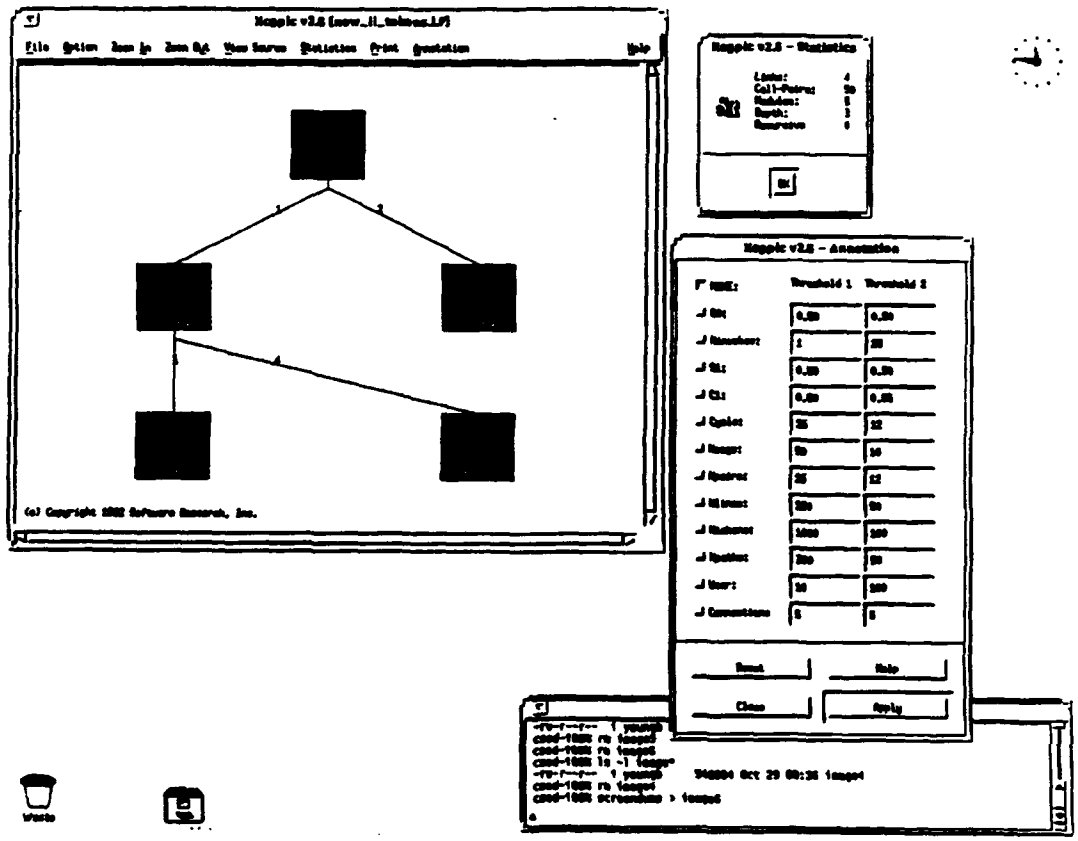


Figure 25-16. S-TCAT/Ada Call Graph for LL_TOKENS

```
Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

Selected SCOVER System Option Settings:
[-c]  Cumulative Report     -- NO
[-p]  Past History Report   -- NO
[-n]  Not Hit Report        -- YES
[-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:   2

S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Call-pair Hit Report.

No.   Module Name:              Call-pair Coverage Status:
1     new_ll_compile.LL_COMPILE       All Call-pairs Hit.  S1 = 100%
2     new_ll_compile.LLMAIN           All Call-pairs Hit.  S1 = 100%
3     new_ll_compile.READGRAM         All Call-pairs Hit.  S1 = 100%
4     new_ll_compile.BUILDRIGHT       All Call-pairs Hit.  S1 = 100%
5     new_ll_compile.BUILDSELECT      All Call-pairs Hit.  S1 = 100%
6     new_ll_compile.PARSE            1    2    3    4    8    9   10
7     new_ll_compile.LLFIND           All Call-pairs Hit.  S1 = 100%
8     new_ll_compile.LLNEXTTOKEN      All Call-pairs Hit.  S1 = 100%
9     new_ll_tokens.ADVANCE           1    2    3    4
10    new_ll_tokens.SCAN_PATTERN      7    8    9   11   17   18   19
                                      22   23   24   28   31   32   33
                                      38   41   42   43
11    new_ll_compile.GET_CHARACTER    All Call-pairs Hit.  S1 = 100%
12    new_ll_tokens.CHAR_ADVANCE      All Call-pairs Hit.  S1 = 100%
13    new_ll_tokens.CURRENT_SYMBOL    All Call-pairs Hit.  S1 = 100%
14    new_ll_compile.MAKE_TOKEN       1    2    3    4    5    6
                                    ...
40    ll_sup_body.EMIT_SCAN_PROC      All Call-pairs Hit.  S1 = 100%
41    ll_sup_body.EMIT_SELECT         1    2    3
42    ll_sup_body.EMIT_CHAR           All Call-pairs Hit.  S1 = 100%
43    ll_sup_body.EMIT_PATTERN_MATCH  5    6    7    8    9   15   16
                                      22   23   24   25
44    ll_sup_body.EMIT_CONCAT_RIGHT   2
45    ll_sup_body.EMIT_CONCAT_CASES   1    2    4    8    9

Number of Call-pairs Hit:              88
Total Number of Call-pairs:           162
S1 Coverage Value:                   54.32%
```

Figure 25-17. S-TCAT/Ada Call-Pair Coverage using test1.lex

```
S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Not Hit Report.

No.   Module Name:              Call-pair Coverage Status:
1     new_ll_compile.LL_COMPILE       All Call-pairs Hit.  S1 = 100%
2     new_ll_compile.LLMAIN           All Call-pairs Hit.  S1 = 100%
3     new_ll_compile.READGRAM         All Call-pairs Hit.  S1 = 100%
4     new_ll_compile.BUILDRIGHT       All Call-pairs Hit.  S1 = 100%
5     new_ll_compile.BUILDSELECT      All Call-pairs Hit.  S1 = 100%
6     new_ll_compile.PARSE            5    6    7   11
7     new_ll_compile.LLFIND           All Call-pairs Hit.  S1 = 100%
8     new_ll_compile.LLNEXTTOKEN      All Call-pairs Hit.  S1 = 100%
9     new_ll_tokens.ADVANCE           5    6
10    new_ll_tokens.SCAN_PATTERN      1    2    3    4    5    6   10
                                      14   15   16   25   26   27   29
                                      36   39   40   44
11    new_ll_compile.GET_CHARACTER    All Call-pairs Hit.  S1 = 100%
12    new_ll_tokens.CHAR_ADVANCE      All Call-pairs Hit.  S1 = 100%
13    new_ll_tokens.CURRENT_SYMBOL    All Call-pairs Hit.  S1 = 100%
14    new_ll_compile.MAKE_TOKEN       7
                                    ...
40    ll_sup_body.EMIT_SCAN_PROC      All Call-pairs Hit.  S1 = 100%
41    ll_sup_body.EMIT_SELECT         4
42    ll_sup_body.EMIT_CHAR           All Call-pairs Hit.  S1 = 100%
43    ll_sup_body.EMIT_PATTERN_MATCH  1    2    3    4   10   11   12
                                      17   18   20
44    ll_sup_body.EMIT_CONCAT_RIGHT   1
45    ll_sup_body.EMIT_CONCAT_CASES   3    5    6    7

Number of Call-pairs Not Hit:          74
Total Number of Call-pairs:           162
S1 Coverage Value:                   54.32%
```

Figure 24-17 continued: S-TCAT/Ada Call-Pair Coverage Using test1.lex

```
Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

Selected SCOVER System Option Settings:
[-c]  Cumulative Report     -- NO
[-p]  Past History Report   -- NO
[-n]  Not Hit Report        -- YES
[-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:   2

S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Call-pair Hit Report.

No.  Module Name:              Call-pair Coverage Status:
1     new_ll_compile.LLFIND          All Call-pairs Hit.  S1 = 100%
2     new_ll_compile.LLPRTSTRING     All Call-pairs Hit.  S1 = 100%
3     new_ll_compile.LLPRTTOKEN
4     new_ll_compile.LLSKIPTOKEN
5     new_ll_compile.LLSKIPNODE
6     new_ll_compile.LLSKIPBOTH
7     new_ll_compile.LLFATAL
8     new_ll_compile.GET_CHARACTER   All Call-pairs Hit.  S1 = 100%
9     new_ll_compile.CVT_STRING      All Call-pairs Hit.  S1 = 100%
10    new_ll_compile.MAKE_TOKEN      1    2    3    4    5    6
11    new_ll_compile.LLNEXTTOKEN     All Call-pairs Hit.  S1 = 100%
12    new_ll_compile.BUILDRIGHT      All Call-pairs Hit.  S1 = 100%
                                  ...
50    ll_sup_body.EMIT_PATTERN_MATCH 5    6    7    8    9   15   16   19
                                     22   23   24   25
51    ll_sup_body.EMIT_CHAR          All Call-pairs Hit.  S1 = 100%
52    ll_sup_body.EMIT_SELECT        1    2    3
53    ll_sup_body.EMIT_SCAN_PROC     All Call-pairs Hit.  S1 = 100%
54    ll_sup_body.EMIT_TOKEN         All Call-pairs Hit.  S1 = 100%
55    ll_sup_body.INCLUDE_PATTERN    1    3    4
56    ll_sup_body.LOOK_AHEAD         All Call-pairs Hit.  S1 = 100%
57    ll_sup_body.LOOK_UP_PATTERN    All Call-pairs Hit.  S1 = 100%
58    ll_sup_body.OPTION             All Call-pairs Hit.  S1 = 100%
59    ll_sup_body.REPEAT             All Call-pairs Hit.  S1 = 100%
60    ll_sup_body.STORE_PATTERN
61    ll_actions.LLTAKEACTION        All Call-pairs Hit.  S1 = 100%

Number of Call-pairs Hit:             88
Total Number of Call-pairs:          253
S1 Coverage Value:                    34.78%
```

Figure 25-18. S-TCAT/Ada Call-Pair Coverage using test1.lex Accounting for All Call-Pairs

```
S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Not Hit Report.

No.  Module Name:              Call-pair Coverage Status:
1     new_ll_compile.LL_IND          All Call-pairs Hit.  S1 = 100%
2     new_ll_compile.LLPRTSTRING     All Call-pairs Hit.  S1 = 100%
3     new_ll_compile.LLPRTTOKEN      1
4     new_ll_compile.LLSKIPTOKEN     1    2
5     new_ll_compile.LLSKIPNODE      1    2
6     new_ll_compile.LLSKIPBOTH      1    2    3
7     new_ll_compile.LLFATAL         1
8     new_ll_compile.GET_CHARACTER   All Call-pairs Hit.  S1 = 100%
9     new_ll_compile.CVT_STRING      All Call-pairs Hit.  S1 = 100%
10    new_ll_compile.MAKE_TOKEN      7
11    new_ll_compile.LLNEXTTOKEN     All Call-pairs Hit.  S1 = 100%
12    new_ll_compile.BUILDRIGHT      All Call-pairs Hit.  S1 = 100%
13    new_ll_compile.BUILDSELECT     All Call-pairs Hit.  S1 = 100%
14    new_ll_compile.READGRAM        All Call-pairs Hit.  S1 = 100%
                                  ...
48    ll_sup_body.EMIT_CONCAT_CASES  3    5    6    7
49    ll_sup_body.EMIT_CONCAT_RIGHT  1
50    ll_sup_body.EMIT_PATTERN_MATCR 1    2    3    4   10   11   12   13   14
                    17   18   20
51    ll_sup_body.EMIT_CHAR          All Call-pairs Hit.  S1 = 100%
52    ll_sup_body.EMIT_SELECT        4
53    ll_sup_body.EMIT_SCAN_PROC     All Call-pairs Hit.  S1 = 100%
54    ll_sup_body.EMIT_TOKEN         All Call-pairs Hit.  S1 = 100%
55    ll_sup_body.INCLUDE_PATTERN    2
56    ll_sup_body.LOOK_AHEAD         All Call-pairs Hit.  S1 = 100%
57    ll_sup_body.LOOK_UP_PATTERN    All Call-pairs Hit.  S1 = 100%
58    ll_sup_body.OPTION             All Call-pairs Hit.  S1 = 100%
59    ll_sup_body.REPEAT             All Call-pairs Hit.  S1 = 100%
60    ll_sup_body.STORE_PATTERN      1
61    ll_actions.LLTAKEACTION        All Call-pairs Hit.  S1 = 100%

Number of Call-pairs Not Hit:        165
Total Number of Call-pairs:          253
S1 Coverage Value:                    34.78%
```

Figure 25-18 continued: S-TCAT/Ada Call-Pair Coverage using test1.lex Accounting for All Call-Pairs

```
Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

Selected SCOVER System Option Settings:
[-c]  Cumulative Report     -- YES
[-p]  Past History Report   -- NO
[-n]  Not Hit Report        -- YES
[-H]  Hit Report            -- YES
[-nh] Newly Hit Report      -- YES
[-nm] Newly Missed Report   -- YES
[-h]  Histogram Report      -- NO
[-l]  Log Scale Histogram   -- YES
[-Z]  Reference Listing S1  -- NO
Options read:   6

S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
```

		Current Test			Cumulative Summary		
Module Name:	Number Of Call-pairs:	No. Of Invokes	Call-pairs Hit	S1% Cover	No. Of Invokes	Call-pairs Hit	S1% Cover
new_ll_compile.LL_COM	1	1	1	100.00	2	1	100.00
new_ll_compile.LLMAIN	2	1	2	100.00	2	2	100.00
new_ll_compile.READGR	2	1	2	100.00	2	2	100.00
new_ll_compile.BUILDR	0	64	0	100.00	128	0	100.00
new_ll_compile.BUILDS	0	64	0	100.00	128	0	100.00
new_ll_compile.PARSE	11	1	7	63.64	2	7	63.64
new_ll_compile.LLFIND	0	312	0	100.00	510	0	100.00
new_ll_compile.LLNEXT	0	221	0	100.00	355	0	100.00
new_ll_tokens.ADVANCE	6	221	5	83.33	355	5	83.33
new_ll_tokens.SCAN_PA	44	455	22	50.00	712	25	56.82
new_ll_compile.GET_CH	0	1385	0	100.00	2250	0	100.00
new_ll_tokens.CHAR_AD	0	1238	0	100.00	2018	0	100.00
new_ll_tokens.CURRENT	0	220	0	100.00	353	0	100.00
new_ll_compile.MAKE_T	7	220	7	100.00	353	7	100.00
new_ll_compile.CVT_ST	0	220	0	100.00	353	0	100.00
new_ll_compile.EXPAND	3	461	1	33.33	715	1	33.33
new_ll_compile.MATCH	0	461	0	100.00	715	0	100.00
new_ll_compile.ERASE	0	707	0	100.00	1105	0	100.00
new_ll_tokens.LOOK_AH	0	84	0	100.00	123	0	100.00
ll_actions.LLTAKEACTI	0	429	0	100.00	659	0	100.00
		...					
ll_sup_body.TAIL	18	2	0	0.00	2	0	0.00
ll_sup_body.EMIT_ALT_	8	12	8	100.00	12	8	100.00
Totals	238	7569	132	55.46	11884	136	57.14

Figure 25-19. S-TCAT/Ada Call-Pair Coverage using test1.lex & sample.lex

```
S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Call-pair Hit Report.

No.  Module Name:                Call-pair Coverage Status:
1     new_ll_compile.LL_COMPILE       All Call-pairs Hit.  S1 = 100%
2     new_ll_compile.LLMAIN           All Call-pairs Hit.  S1 = 100%
3     new_ll_compile.READGRAM         All Call-pairs Hit.  S1 = 100%
4     new_ll_compile.BUILDRIGHT       All Call-pairs Hit.  S1 = 100%
5     new_ll_compile.BUILDSELECT      All Call-pairs Hit.  S1 = 100%
6     new_ll_compile.PARSE            1    2    3    4    8    9   10
7     new_ll_compile.LLFIND           All Call-pairs Hit.  S1 = 100%
8     new_ll_compile.LLNEXTTOKEN      All Call-pairs Hit.  S1 = 100%
9     new_ll_tokens.ADVANCE           1    2    3    4    5
10    new_ll_tokens.SCAN_PATTERN      1    2    3    7    8    9   11   17   18
                                      19   20   21   22   23   24   28   31    3
                                      33   34   37   38   41   42   43
11    new_ll_compile.GET_CHARACTER    All Call-pairs Hit.  S1 = 100%
                                    ...
48    ll_sup_body.RESOLVE_AMBIGUITY   1    2    9   10   11   12   13   27   28
                                      29   34   35   36
49    ll_sup_body.RESTRICT            1    2    3    4    5    6    7
50    ll_sup_body.TAIL
51    ll_sup_body.EMIT_ALT_CASES      All Call-pairs Hit.  S1 = 100%

Number of Call-pairs Hit:              136
Total Number of Call-pairs:            238
S1 Coverage Value:                     57.14%


S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Call-pair Newly Hit Report.

No.  Module Name:                Call-pair Coverage Status:
9     new_ll_tokens.ADVANCE           5
10    new_ll_tokens.SCAN_PATTERN      1    2    3
                                    ...
48    ll_sup_body.RESOLVE_AMBIGUITY   1    2    9   10   11   12   13   27   28
                                      29   34   35   36
49    ll_sup_body.RESTRICT            1    2    3    4    5    6    7
51    ll_sup_body.EMIT_ALT_CASES      1    2    3    4    5    6    7    8
```

Figure 25-19 continued: S-TCAT/Ada Call-Pair Coverage using test1.lex & sample.lex

```
S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Not Hit Report.

No.  Module Name:                Call-pair Coverage Status:
1     new_ll_compile.LL_COMPILE       All Call-pairs Hit.  S1 = 100%
2     new_ll_compile.LLMAIN           All Call-pairs Hit.  S1 = 100%
3     new_ll_compile.READGRAM         All Call-pairs Hit.  S1 = 100%
4     new_ll_compile.BUILDRIGHT       All Call-pairs Hit.  S1 = 100%
5     new_ll_compile.BUILDSELECT      All Call-pairs Hit.  S1 = 100%
6     new_ll_compile.PARSE            5    6    7    11
7     new_ll_compile.LLFIND           All Call-pairs Hit.  S1 = 100%
8     new_ll_compile.LLNEXTTOKEN      All Call-pairs Hit.  S1 = 100%
9     new_ll_tokens.ADVANCE           6
10    new_ll_tokens.SCAN_PATTERN      4    5    6    10   12   13   14   15   16
                                      25   26   27   29   30   35   36   39   4
                                      44
11    new_ll_compile.GET_CHARACTER    All Call-pairs Hit.  S1 = 100%
                                    ...
40    ll_sup_body.EMIT_SCAN_PROC      All Call-pairs Hit.  S1 = 100%
41    ll_sup_body.EMIT_SELECT         4
42    ll_sup_body.EMIT_CHAR           All Call-pairs Hit.  S1 = 100%
43    ll_sup_body.EMIT_PATTERN_MATCH  2   11   12   13   14   17   18
44    ll_sup_body.EMIT_CONCAT_RIGHT   All Call-pairs Hit.  S1 = 100%
45    ll_sup_body.EMIT_CONCAT_CASES   3    5
46    ll_sup_body.CVT_STRING          All Call-pairs Hit.  S1 = 100%
47    ll_sup_body.CVT_ASCII           All Call-pairs Hit.  S1 = 100%
48    ll_sup_body.RESOLVE_AMBIGUITY   3    4    5    6    7    8    14   15   16
                                      17   18   19   20   21   22   23   24   2
                                      26   30   31   32   33
49    ll_sup_body.RESTRICT            8    9    10   11   12   13
50    ll_sup_body.TAIL                1    2    3    4    5    6    7    8    9
                                      10   11   12   13   14   15   16   17   1
51    ll_sup_body.EMIT_ALT_CASES      All Call-pairs Hit.  S1 = 100%

Number of Call-pairs Not Hit:          102
Total Number of Call-pairs:            238
S1 Coverage Value:                      57.14%


S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.

S1 Call-pair Newly Missed Report.

No.  Module Name:                Call-pair Coverage Status:
10    new_ll_tokens.SCAN_PATTERN      19   20   21
40    ll_sup_body.EMIT_SCAN_PROC      3
```

Figure 25-19 continued: S-TCAT/Ada Call-Pair Coverage using test1.lex & sample.lex

```
S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
Call-pair Level Histogram for Module: new_ll_compile.LL_COMPILE

                                +-------------------------------------------------+
                                | Logarithm of Executions, Normalized to Maximum
                                | (Maximum =      2 Hits)
 Call-pair Number Of            |
 Number    Executions >----------1----------10-----20----30---40--80-100
+---------------------+-------------------------------------------------+
                      |
    1             2   | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                      |
+---------------------+-------------------------------------------------+
Average Hits per Executed Call-pair:   2.0000
S1 Value for this Module:            100.0000
                                ...
S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
Call-pair Level Histogram for Module: new_ll_compile.BUILDSELECT

      No call-pairs present or hit

S-TCAT/C: Coverage Analyzer.  [Release 8.2 for SUN/UNIX 09/16/92]
(c) Copyright 1990 by Software Research, Inc.
Call-pair Level Histogram for Module: new_ll_compile.PARSE

                                +-------------------------------------------------+
                                | Logarithm of Executions, Normalized to Maximum
                                | (Maximum =   1105 Hits)
 Call-pair Number Of            |
 Number    Executions >----------1----------10-----20----30---40--80-100
+---------------------+-------------------------------------------------+
                      |
    1             2   | XXXXXXX
    2             2   | XXXXXXX
    3             2   | XXXXXXX
    4           353   | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    5   *             |
    6   *             |
    7   *             |
    8           715   | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    9           659   | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   10          1105   | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   11   *             |
                      |
+---------------------+-------------------------------------------------+
        (* = Zero Hits)
Average Hits per Executed Call-pair: 405.4286
S1 Value for this Module:             63.6364
                                ...
```

Figure 25-19 continued: S-TCAT/Ada Call-Pair Coverage using test1.lex & sample.lex

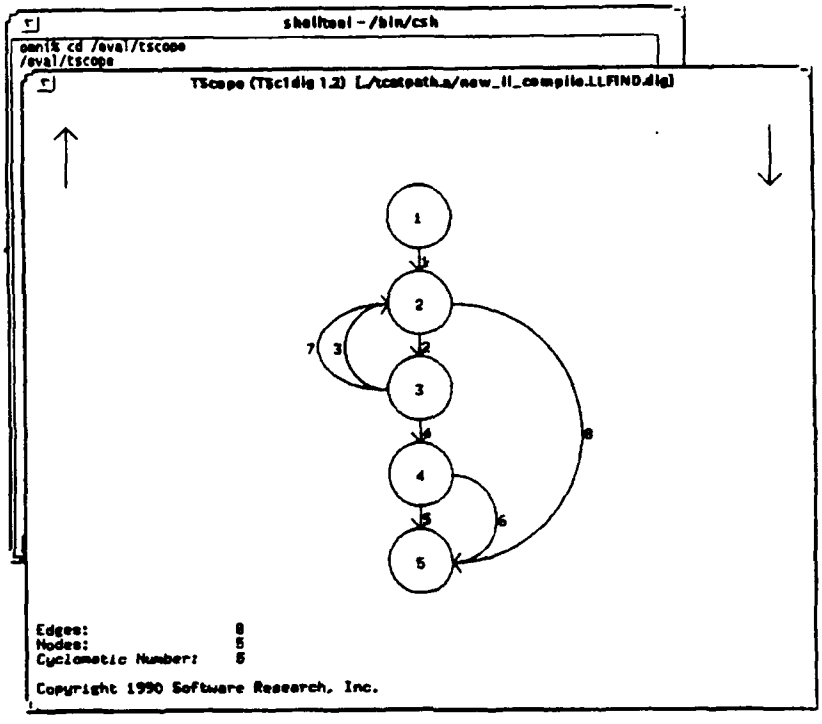


Figure 25-20. TSCOPE Dynamic Display of Coverage on Directed Graph for LLFIND

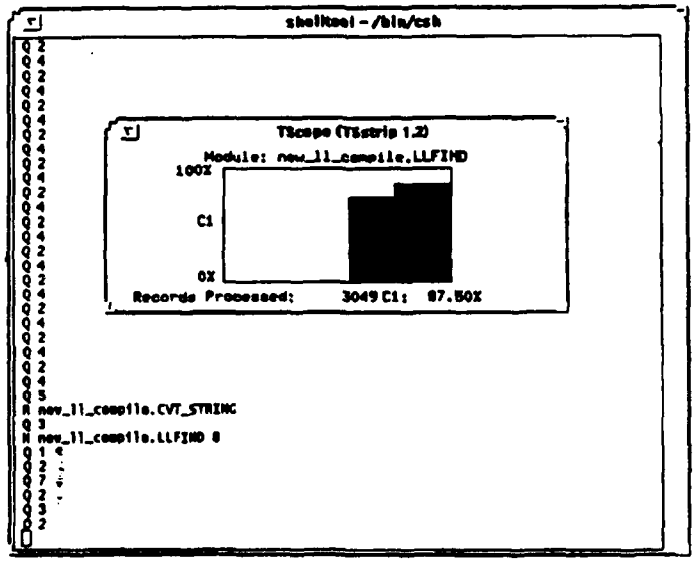


Figure 25-21. TSCOPE Dynamic Display of Coverage Accumulation for LLFIND

```
{%c Values file 1: for variable number of initial TDGen executions. }

expr         {% expr}{% op}{% expr}  {% identifier}  {% real_no}
op           + - / *
identifier   variable1 variable2 {% alpha 6}
real_no      {% real 4.6} {% integ1}E+{% integ2} {% integ1}E-{% integ2}
integ1       {%r 1..100}
integ2       {%r 3..6}


{%c Values file 2: for last two executions of TDGen. }

expr         variable1 variable2 {% alpha 6}              {% real 4.6} {% integ1}E+{% i
op           + - / *
identifier   variable1 variable2 {% alpha 6}
real_no      {% real 4.6} {% integ1}E+{% integ2} {% integ1}E-{% integ2}
integ1       {%r 1..100}
integ2       {%r 3..6}


{%c Template file: Produces arithmetic expression of varying    }
{%c complexity for use in testing a generated lexical analyzer. }

{% expr }
```

Figure 25-22. TDGen Sample Value and Template Files

Field	No. Table Entries	Cumulative Total Combinations
% expr	3	3
% op	4	12
% identifier	3	36
% real_no	3	108
% integ1	100	10800
% integ2	4	43200

Figure 25-23. TDGen Table of Sequential Combinations for Initial Files

```
{% real_no}
{% expr}{% op}{% expr}
{% identifier}
{% expr}{% op}{% expr}
{% identifier}
{% identifier}
{% expr}{% op}{% expr}
{% real_no}
{% real_no}
{% identifier}
```

Figure 25-24. TDGen Output of First Random Execution

```
3E+6
{% real 4.6}-variable1
RSBEz4
{% integ1}E-{% integ2}-{% integ1}E-{% integ2}
variable2
variable2
{% identifier}*{% real_no}/{% identifier}/{% identifier}
21E+4
47E-6
variable1
```

Figure 25-25. TDGen Output After 3 Executions with 1st Value File

```
3E+6
3092.703258-variable1
RSBEz4
53E-4-83E-6
variable2
variable2
G36dk5*26E-5/c1mHEJ/variable2
21E+4
47E-6
variable1
```

Figure 25-26. TDGen Output After 2 Executions with 2nd Value File

26. TST

The Ada Test Support Tool (TST) is government owned. Designed to facilitate the testing of Ada subprograms and task entry points, it provides test driver generation with test data generation for program unit parameters. TST operates in batch or interactive mode.

26.1 Tool Overview

TST was developed by Intermetrics, Inc. under contract to the Software Technology for Adaptable, Reliable Systems (STARS) Foundations program. It leverages off technology developed for the Ada Test and Analysis Tools (ATEST) Intermetrics previously built for the Worldwide Military Command and Control (WWMCCS) Information System (WIS) program. The first version of this toolset became available in 1989. It is compiler independent and designed to be portable. Intermetrics has hosted TST on the Alsys PC/AT, Alsys Sun, and DEC Ada compilers. It is available at no charge from the STARS Foundation Archive.

The evaluation was performed on TST version 2 running in a VAX/VMS environment.

TST consists of three parts: a Shell, Source Instrumentor, and Testing Subsystem. When used interactively, the Shell provides a test environment where the user can set various default parameters, such as the name of a separate directory to hold all the files generated during instrumentation. It allows the user to invoke the Source Instrumentor and handle the compilation, linking, and execution of the instrumented code. The Shell also provides for the management of internal TST files, and screen and terminal handling.

Testing starts by invoking the Source Instrumentor to insert statements that allow calling contained subprograms and task entry points into a library unit under test. The instrumentation caters for reading and writing of parameter values, assertion testing, and logging of test results and program execution information. Both the specification and body of the unit(s) under test are submitted to the Source Instrumentor. (Although statements are not inserted in package specifications, the instrumentor does extract some information from them.) Units containing type declarations that are used by the unit under test are also needed. The user is required to submit additional information for testing generic items. For example, instrumentation of a generic package requires the user to provide actual type and subprogram names, whereas a generic formal type or subprogram requires a package name and then the name of the actual type or subprogram. In the case of generic declarations, the

user must provide type and subprogram names for generic parameters. One instantiation of each generic unit is generated using specified names.

Statements for automatic test data generation for predefined parameter types are automatically included in the source code. The user is queried whether test data generation for user-defined types should be included. Test data generation is performed in two ways. In one case, TST generates all possible values for a parameter (or the first and last values for floating point types). In the other, the user specifies that all possible values are divided into a given number of partitions, and TST then selects the first, middle, and last value from each partition. The user is responsible for ensuring that the number of values generated is not sufficient to cause the Ada exception *Storage_Error* to be raised. This automatic test data generation is not available for task, private, or limited private types. When requesting generation for unconstrained types, such as an unconstrained array, record, or string, the user must give a constraint. Optionally, constraints may also be given for character types.

Finally, the Source Instrumentor generates a test driver to call the routines contained in the library unit. This test driver is included at the end of the instrumented source file. At the user's option, the Source Instrumentor also prepares a pretty printed source code listing. This listing includes breakpoint numbers that are used in path analysis reports to identify the statements that were executed.

Once the generated testbed has been compiled and linked, it can be invoked under TST and then TST hands control to the Testing Subsystem. This subsystem provides a dual-window user interface. The user interacts with TST through the Dialogue Window, while the Display Window provides useful information in the form of declarations for all the routines that may be tested and for current assertions. (A TST option provides for handling data output to the screen from the unit under test. This option allows, for example, directing the unit output to the Testing Subsystem windows, or to another window superimposed over these.)

The user is asked for a test identification, and the name of the Test Data File (TDF) that contains the assertions and calls that will be used to test the unit. If the named TDF does not exist, the Testing Subsystem saves the user's subsequent test input in the named file so that a test run can be easily repeated. TDFs can also be created or modified outside of TST. Testing proceeds by calling procedures, functions, and entry points within the unit under test and making assertions about the output. Each routine is identified using the numbers given in the display window and, when the user requests its call, TST queries for input parameter values. He may enter actual parameter values using named or positional notation. Alternatively, the user may request automatic test data generation for a parameter. The user

must also specify values for OUT mode parameters. These will be used for constraints where required, for example, for string OUT parameters. The Testing Subsystem calls the associated routine for each generated combination of test data. Once a test is complete, the values of OUT and IN OUT parameters, or function results, are displayed in the Dialogue Window.

Assertions can be given to check these test results. These assertions may be global, that is, valid from the time the assertion is given until either the end of the test session or the deletion of that assertion. Alternatively, local assertions are valid only for the next call command, or the multiple calls of a single routine that may be incurred by test data generation. The validity of assertions is not checked when defined, but only when an assertion is evaluated for the specified results. When an assertion fails, a message is printed to the screen and the testbed will either continue, abort and start report generation, or query the user whether to continue or abort depending on how an *Assertion_Handling* flag is set.

The user can give a number of other commands to the Test Subsystem. These are used, for example, to control the display area, manage assertions, and control the handling of any screen output generated by the routine under test.

TST automatically generates a TST report at the end of a test execution. As well as general identification information, this report lists the Ada declarations for all visible procedures, subprograms, and entry points of the unit under test, test data that was generated, and results of invoked routines and associated assertions. The user may request a path analysis report to be included. This report provides a trace of the execution history and an execution summary report that lists the number of times each statement (or group of consecutive statements) was executed.

26.2 Observations

Ease of use. The on-line help provides summaries of Shell and Testing Subsystem commands that are very helpful. A simple "?" provides a list of currently available commands.

Limited tailorability is available. The help file format is tailorable, allowing the user to modifying existing messages or add new messages. The user can define the type of terminal being used via an ANSI X3.64 Compatible Virtual Terminal Package Terminal capabilities files, a variation of the TERMCAP developed in the Berkeley extensions to Unix. This allows, for example, user-defined function keys.

Documentation and user support. The installation instructions received with the software from the STARS Foundation Repository had some minor omissions. The TST documentation itself is easy to follow and helpful. No direct support for the tool is available, although Intermetrics answered questions that arose during this examination.

Instrumentation overhead. The instrumentation performed by TST imposes a substantial overhead. The degree of code expansion largely depends on the number and type of type definitions encountered, and number of subprogram units being tested. In the case of the Ada Lexical Analyzer Generator, the source code size of a library routine containing the function LLFIND alone was four blocks. When instrumented for full test data generation, this size increased to 124 blocks.

However, since bottom-up testing requires units to be tested independently or in small groups, with careful partitioning of the code, the instrumentation overhead may not be a significant problem.

Ada restrictions. The generated control program is subject to the same restrictions that any program would be in calling package subprograms. For example, values cannot be given to, or received from, objects declared as private. TST imposes additional restrictions largely to do with the format and content of input data to the subprograms being tested. These restriction include the following: (1) values must be given for all parameters that do not have defaults and named notation is required for use of defaults; (2) all parameter values and assertion values must be literal values, and (3) test data generation is not supported for tasks types, private types, limited private types, or records types with nested variant parts.

Problems encountered. A couple of problems during instrumentation required minor manual editing before the instrumented source code would compile. After instrumentation, some Ada *use* statements had to be manually inserted to cater for inserted *with* statement. Some problems were experienced generating test data for string subtypes.

26.3 Recent Changes

Intermetrics has continued development of TST. The augmented version is, however, a proprietary Intermetrics tool.

26.4 Sample Outputs

Figures 26-1 through 26-6 provide sample outputs from TST.

```
Test Support Tool - Version 2.0                    Page: 1
Test Configuration Report
-------------------------------------------------------------

      Program Under Test:       LL_COMPILE3

      Test Date:                09/03/92

      Test Day:                 THURSDAY

      Test Time:                12:57:50

      Data File:                RUN3.TDF

      Test ID:                  Test LLFIND Run 3

      Executable File:          Name = NOT AVAILABLE
                                Date = ????????
                                Time = ????????

      Defaults:                 TST_DIR = [ADATEST.TST.TSTDIR]
                                RPT_CPL = 60
                                RPT_LPP = 54
                                ASSERTION_HANDLING = CONTINUE
                                SCREEN_ECHO = ON
```

```
Test Support Tool - Version 2.0                    Page: 2
Test Configuration Report
-------------------------------------------------------------

    1     function LLFIND(
            ITEM : LLSTRINGS;
            WHICH : LLSTYLE) return INTEGER;

    2     procedure LLMAIN;
```

Figure 26-1. TST Test Configuration File for Function LLFIND

```
Test Support Tool - Version 2.0                    Page: 3
Parameter Report

================================================================
GLOBAL Assertion  1) 1 < 33
================================================================
Unit Under Test: ( 2)

procedure LLMAIN;

================================================================
LOCAL Assertion  1) 1 = 12
Unit Under Test: ( 1)

function LLFIND(
    ITEM : LLSTRINGS;
    WHICH : LLSTYLE) return INTEGER;

  Parameter          Entering Value     Exiting Value
  ------------------ ------------------ --------------------
  ITEM               "Identifier  ..."
  WHICH              GROUP
  <RETURN_VALUE>                        12
================================================================
Unit Under Test: ( 1)

function LLFIND(
    ITEM : LLSTRINGS;
    WHICH : LLSTYLE) return INTEGER;

Test Data Automatically Generated
WHICH => *
  Parameter          Entering Value     Exiting Value
  ------------------ ------------------ --------------------
  ITEM               "ASCII       ..."
  WHICH              LITERAL
  <RETURN_VALUE>                        10
 ................................................................
  ITEM               "ASCII       ..."
  WHICH              NONTERMINAL
  <RETURN_VALUE>                        0
 ................................................................
  ITEM               "ASCII       ..."
  WHICH              GROUP
  <RETURN_VALUE>                        0
 ................................................................
  ITEM               "ASCII       ..."
  WHICH              ACTION
```

Figure 26-2. TST Parameter Report for Function LLFIND

```
  <RETURN_VALUE>                          0
 ..........................................................
  ITEM                 "ASCII        ..."
  WHICH                PATCH
  <RETURN_VALUE>                          0
==========================================================
LOCAL Assertion  2) 1 = 0
Unit Under Test: ( 1)

function LLFIND(
    ITEM : LLSTRINGS;
    WHICH : LLSTYLE) return INTEGER;

Test Data Automatically Generated
WHICH => *
  Parameter            Entering Value      Exiting Value
  -------------------- ------------------- --------------------
  ITEM                 "                ..."
  WHICH                LITERAL
  <RETURN_VALUE>                          0
 ..........................................................
  ITEM                 "                ..."
  WHICH                NONTERMINAL
  <RETURN_VALUE>                          0
 ..........................................................
  ITEM                 "                ..."
  WHICH                GROUP
  <RETURN_VALUE>                          0
 ..........................................................
  ITEM                 "                ..."
  WHICH                ACTION
  <RETURN_VALUE>                          0
 ..........................................................
  ITEM                 "                ..."
  WHICH                PATCH
  <RETURN_VALUE>                          0
==========================================================
LOCAL Assertion  3) 1 = 12
Unit Under Test: ( 1)

function LLFIND(
    ITEM : LLSTRINGS;
    WHICH : LLSTYLE) return INTEGER;

  Parameter            Entering Value      Exiting Value
  -------------------- ------------------- --------------------
  ITEM                 "ASCII        ..."
  WHICH                LITERAL
  <RETURN_VALUE>                          10

*** LOCAL Assertion  3)  1  = 12 Failed
```

Figure 26-2 continued: TST Parameter Report for Function LLFIND

```
Test Support Tool - Version 2.0                    Page: 6
Execution History Report
------------------------------------------------------------

Begin LL_COMPILE3.LLMAIN.READGRAM.BUILDRIGHT
      [77]
Begin LL_COMPILE3.LLMAIN.READGRAM.BUILDSELECT
      [52-55][55][55][55][55][55][55][55][55][55][55][55]
      [55][55][55][55][55][55][55][55-58][60] [55]
                                ...
End LL_COMPILE3.LLMAIN.READGRAM.BUILDSELECT
Resume LL_COMPILE3.LLMAIN.READGRAM.BUILDRIGHT
      [78]
End LL_COMPILE3.LLMAIN.READGRAM.BUILDRIGHT
Begin LL_COMPILE3.LLFIND
      [1-6] [10]  [10] [7-8]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6] [10]   [7-8]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6] [10]   [7][9]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6] [10]   [7][9]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6] [10]   [7][9]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6] [10]   [7][9]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6]     [11]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6]     [11]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6]     [11]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6]     [11]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6]     [11]
End LL_COMPILE3.LLFIND
Begin LL_COMPILE3.LLFIND
      [1-6] [10]   [7-8]
End LL_COMPILE3.LLFIND
```

Figure 26-3. TST Execution History Report for Function LLFIND

```
Test Support Tool - Version 2.0                    Page: 21
Execution Summary Report
-------------------------------------------------------------

      Statement              Execution Count
      ---------              ---------------

LL_COMPILE3
      [1-3]                          12
      [4-5]                          65
      [6]                            50
      [7]                             7
      [8]                             3
      [9]                             4
      [10]                            8
      [11]                            5
      [12]                            0
      [13-15]                        64
      [16-19]                       174
      [20-23]                        50
      [24]                           61
      [25-28]                        46
      [29-32]                        13
      [33]                            4
      [34-35]                         0
      [36]                          174
      [37]                           79
      [38]                           95
      [39]                          174
      [40]                          144
      [41]                           30
      [42]                          174
      [43-44]                         0
      [45-47]                        64
      [48-49]                       227
      [50-51]                        64
      [52-53]                         1
      [54]                           32
      [55]                          640
      [56-58]                        32
      [59]                            6
      [60]                           26
      [61-64]                         1
      [65-70]                        64
      [71]                            1
      [72-74]                        26
      [75-78]                         1
```

Figure 26-4. TST Execution Summary Report for Function LLFIND

```
GLOBAL_ASSERT( 1 < 33    )
--* 2
--* 1
LOCAL_ASSERT(    1  = 12 )
CALL_ROUTINE  1 (
ITEM => "Identifier          "
WHICH => GROUP
);
--* 1
CALL_ROUTINE  1 (
ITEM => "ASCII               "
WHICH => LITERAL
);

                                                    ...
CALL_ROUTINE  1 (
ITEM => "ASCII               "
WHICH => PATCH
);
--* 1
LOCAL_ASSERT(    1  = 0 )
CALL_ROUTINE  1 (
ITEM => "                    "
WHICH => LITERAL
);
LOCAL_ASSERT(    1  = 0 )
CALL_ROUTINE  1 (
ITEM => "                    "
WHICH => NONTERMINAL
);
LOCAL_ASSERT(    1  = 0 )
CALL_ROUTINE  1 (
ITEM => "                    "
WHICH => GROUP
);
LOCAL_ASSERT(    1  = 0 )
CALL_ROUTINE  1 (
ITEM => "                    "
WHICH => ACTION
);
LOCAL_ASSERT(    1  = 0 )
CALL_ROUTINE  1 (
ITEM => "                    "
WHICH => PATCH
);
--* 1
LOCAL_ASSERT(    1  = 12 )
CALL_ROUTINE  1 (
ITEM => "ASCII               "
WHICH => LITERAL
);
```

Figure 26-5. TST Sample Test Data File for Function LLFIND

```
with  LL_DECLARATIONS, INTEGER_TEXT_IO, TEXT_IO;
package LL_COMPILE3  is
use  LL_DECLARATIONS, INTEGER_TEXT_IO, TEXT_IO;

PARSING_ERROR: exception;    --  for fatal parsing errors
type  LLSTYLE  is  (LITERAL, NONTERMINAL, GROUP, ACTION, PATCH);
type  LLSYMTABENTRY  is      -- for symbol table entries
   record
      KEY: LLSTRINGS;    -- literal string or group identifier
      KIND: LLSTYLE;     -- literal or group
   end record;
                                        ...
LLSYMBOLTABLE: array ( 1 .. LLTABLESIZE ) of LLSYMTABENTRY;
                        -- the symbol table for literal terms
function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER;
procedure  LLMAIN;
end LL_COMPILE3;

with LL_DECLARATIONS, INTEGER_TEXT_IO, TEXT_IO;
package body LL_COMPILE3 is
use LL_DECLARATIONS, INTEGER_TEXT_IO, TEXT_IO;

function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
   --  Find item in symbol table -- return index or 0 if not found.
   --  Assumes symbol table is sorted in ascending order.
   LOW, MIDPOINT, HIGH: INTEGER;
begin
   LOW := 1;
   HIGH := LLTABLESIZE + 1;
   while  LOW /= HIGH  loop
      MIDPOINT := (HIGH + LOW) / 2;
      if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
         HIGH := MIDPOINT;
      elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
         if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
            return( MIDPOINT );
         else
            return( 0 );
         end if;
      else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
         LOW := MIDPOINT + 1;
      end if;
   end loop;
   return( 0 );  -- item is not in table
end  LLFIND;

procedure  LLMAIN  is
                                        ...
end  LLMAIN;
end LL_COMPILE3;
```

Figure 26-6. TST Function LLFIND

27. Test/Cycle and Metrics Manager

Test/Cycle supports the definition of functional requirements and validation criteria such as test plans, test runs, and test cases. It provides users with a graphical object-oriented framework for developing test plans, managing of software testing efforts, and problem reporting. The concept of *software builds* supports incremental development.

Metrics Manager is a measurement system that focuses on productivity and quality improvement. It is supported with an industry data base of software metrics that allows Metrics Manager users to assess their productivity in relation to other organizations. Currently with data from over two hundred projects from some twenty five Fortune 500 organizations, this database is expected to double in size in the near future.

27.1 Tool Overview

Test/Cycle and Metrics Manager are marketed by Computer Power Group, Inc. This organization primarily markets testing services, consulting, and training in software testing and quality assurance. It is a founding member of the Quality Assurance Institute and still serves as a board member. Computer Power Group continues to undertake research into software quality measurement in conjunction with the Quality Assurance Institute and other groups.

Test/Cycle was released in 1990 and is used at some 10 to 15 sites. Metrics Manager first became available in 1989 and is used at 30 to 35 sites. Both tools are PC based and run under MS-DOS; Test/Cycle is available under Microsoft Windows. Both tools are available under IBM's Ad/Cycle tool set. Metrics Manager can import project data from Project Workbench; it also supports a bidirectional interface to Project Bridge for exchange of function point data. At the time of examination, the price of Test/Cycle started at $3,500. Metrics Manager is now marketed by ABT Technologies and its price currently starts at $14,950; this includes a consulting project to set up appropriate metrics for the client. The examinations were performed on Test/Cycle version 3.02 and Metrics Manager version 2.02.

27.1.1 Test/Cycle Overview

Test/Cycle supports Computer Power Group's Testing Management Methodology. The underlying test model is based on the following object types:

- Project. The collection of all data associated with a system under test.
- Requirement. A functional specification of what an application must do.
- Build. A build is a functionally independent group of modules that supports a well-defined system function or a small logical subset of a system.
- Test plan. Initially records the overall testing strategy, test case design, and test case execution. Subsequently includes information on the builds, test runs, and test cases associated with the strategy, and records the test execution results.
- Test run. A series of related test cases combined to test specific requirements, or to perform a specific category of testing.
- Test case. Consists of the description of a program's input data and expected output, together with a description of the steps needed to execute the test case.
- Test file. An object containing information about the actual data that is used during execution of tests.
- Component. A product of the development process, primarily program names, source code names, and libraries supported by a description such as what it contains, how it is created, and who is responsible for maintenance.

Links are used to define the relationships between instances of these objects. While each test case is automatically linked to a single test run, the user can define the desired links between other objects. (The link between test runs and test cases is the only non-commutative link; it is established from the test case perspective so that every test case must belong to a test run.) Test/Cycle does impose some rules that guide the definition of legal links; for example, only leaf requirements may be linked to a test case.

A user starts by defining a project in terms of a unique identifier, organizational information, and narrative ASCII text; similar information is captured for all object types. The project description may be accompanied by characteristics that subsequently can be used to classify both requirements and test cases.

Typically, the next step is to define the requirements that will be used to drive test planning. An initial high-level requirement is defined and successively refined, building a hierarchical tree-like structure of progressively more detailed requirements. Each requirement is automatically assigned a hierarchy level number that indicates its relative position within the hierarchy. A special flag is used to indicate whether testing of a requirement is required. A status of testing not required invokes special handling; for example, the user must give a reason, and these exceptional cases cannot be linked to a test case or a test run.

The user then proceeds to describe the other necessary objects. In addition to the basic descriptive information, each type of object requires some special data. Builds, for example, form a logical grouping of test runs and may require a build group number and group sequence number to indicate build sequencing. Additionally, a test level attribute represents the test level (integration, acceptance, or released) achieved by a build. Test runs are accompanied by a test log that identifies the person responsible for the testing and maintains a record of test events, and test run sequencing information that indicates dependencies between test runs. Test cases include information on test set up, the tester, and one or more description steps. Each test case description step includes the results expected, a pass or fail indicator, and failure action. Test/Cycle distinguishes between three types of test plan: unit, integration, acceptance. In each case, the narrative description is given according to a preformatted outline, and is accompanied by attributes and the actual test plan definition. Test files are supported by capturing the record type for records contained in the test file. Finally, components have a distinguishing predefined type such as a program type.

Based on its placement in the hierarchy and association with other requirements, a requirement may be subject to one of three levels of testing: high, intermediate, and detail. Test/Cycle generates a separate validation matrix for each level. In the High Level Validation Matrix requirements are cross-referenced to builds, in the Intermediate Level Validation Matrix to test runs, and in the Detail Level Validation Matrix to test cases. These matrices show both explicit and implicit links. (An implicit link is one created when a requirement in the subtree of a higher level requirement is directly linked to a build, test run, or test case.) One of their primary purposes is to show which test cases test which requirements and thus provide insight into test completeness. Test/Cycle measures requirements coverage as the percentage of requirements that are validated by a set of test cases. Based on user entries, Test/Cycle also tracks the number of times a test run is executed. This allows reporting on the number and percent of test steps that have executed successfully, with the date and time a test case was 100% validated. In addition to ensuring that all requirements are tested and all test cases are used, these matrices provide additional types of information. The cross-reference between requirements and components, for example, helps to identify the parts of a system affected by a requirement change and the tests that need to be rerun.

Test/Cycle provides a range of off-line reports and several on-line reports. Off-line reports are available to provide descriptions of the existing objects of each type. On-line displays provide various status information, for example, the status of each test case linked to

a leaf requirements and of individual test cases in the test plan. Additionally, an error/exception status report gives a series of eleven consistency checks of Test/Cycle maintained data, for example, test cases with no requirements linked. Some of these checks employ user-defined threshold values, such as the number of leaf requirements with more than an acceptable number of test cases linked. Progress reporting is provided for each object. In each case, an on-line validation status report summarizes the status of test cases or runs, as appropriate, linked to that component. This information includes the number (and percentage) of test cases, or runs, that have passed, and cumulative validation statistics for the object in question.

Test/Cycle reports and tracks Work Requests (WRs). These can be classified as *problem*, *change*, or *other* (other questions or discrepancies) requests. While roughly similar information is captured for problems and changes, less is captured for other requests. For example, statistical information is only kept for problems and changes; details are kept on problem insertion and discovery, a characterization, and cost to fix data (for problems) or cost to implement data (for changes). Problem and change requests capture information about the phase and activity when a problem was introduced and when identified; this provides some support for continual process improvement. They allow five priority levels, three severity levels, and distinguish between five different classifications.

For WRs in general, sequence numbers are automatically assigned to support an audit trail. WRs are treated as an object type and, consequently, they can be linked to instances of any other object type. A checklist is maintained showing the status of each WR. Predefined reports are available to provide descriptions of individual WRs and a WR log.

27.1.2 Metrics Manager Overview

Metrics Manager supports the collection and analysis of quality and productivity metrics for management purposes. Data can be collected on a monthly, quarterly, or annual basis to monitor the performance of an organization and track the impact of new methods, organizational structures, and technologies. The user starts by modeling the MIS function. The highest level of structure is called the Enterprise. An Enterprise consists of MIS Departments, each of which has a number of products. For metrics reporting and graphing purposes, products can be defined as members of an overall application or system, yielding a composite product called an Application. Aggregates are a special subset of an application

which combine products that are not part of a single application; they allow, for example, reporting on all enhancements regardless of product identification.

Data is stored in a Basic Operating Database (BOD) which captures the lowest level data for each enterprise, each MIS department within the enterprise, and for each product within each MIS department. This includes quality, productivity, cycle time, cost, size, scope, and reliability data as well as indicative and descriptive data used for comparison and categorization purposes. Over three hundred measures and attributes are captured. Once a set of basic data items has been entered in the BOD and validated, the quality and productivity metrics can be computed. These are computed using data up to a user-defined Period Ending Date, and only for products whose development cycle has ended by this date. Subsequently database reports are available to provide information at the enterprise, MIS, and product levels.

The computed data can also be used to produce a range of graphs depicting the enterprise performance. The user selects *x*- and *y*-axes from sets of predefined options and, optionally, one of a number of predefined filters that select a subset of data for graphing. Up to eight graph definitions can be stored for subsequent reuse. While basic metrics graphs operate against enterprise data, the user can request graphs that compare the data from an *Enterprise to the data from the industry database.* (The industry database is a regularly updated, statistically validated database that has been built from client data.) The graphing function also allows each enterprise to define a Critical Metric Set (CMS), that is, those metrics that have been determined to be key management factors in determining enterprise success. Graphs are accompanied by a graph audit report that prints the values from each product that are used to develop each graph. This is intended for use in understanding the metrics calculations and in comparing the same attribute or metric across all applicable products. Graphs are displayed on the screen, but may be sent to a file for subsequent printing on a dot-matrix or laser printer.

Finally, a merge/extract function is provided for those cases where multiple copies of Metrics Manager are installed across an organization. It is used to consolidate data into a single database for analysis and reporting purposes.

27.2 Observations

Ease of use. These tools use an object-oriented, graphical, Common User Access (CUA) user interface to facilitate tool use. Both interfaces are menu driven with context-sensitive on-line help and an analog box of information for selections where appropriate.

The user can use a mouse to select objects in different ways: "point and click," double click, or, unique to Test/Cycle, "drag and drop" which represents direct manipulation of objects; accelerator keys are provided as an alternative method of user interaction. Two types of editing are available. The first is a limited edit using keyboard editing keys such as insert, backspace, home. The second type available through the zoom narrative button is a full edit which allows use of the Windows Clipboard Editing facility. The search filters and user-tailorable project characteristics provide a basis for on-line searching of the test data base. Test/Cycle also supports on-line browsing of the links between objects. Test/Cycle supports only limited customization: the test plan format can be modified by editing the test plan narrative file to include a desired outline.

Documentation and user support. The documentation for these tools is easy to follow and complete.

Problems encountered. *No problems were encountered in the use of these tools.*

27.3 Planned Additions

A new version of Metrics Manager is due for release in late 1992. This new release will include additional functionality. For example, Metrics Manager will help a user to determine the scope of impact of a proposed requirements change, and allow reporting on this maintenance effort independently from the rest of the system. Both Test/Cycle and Metrics Manager will be integrated with ABT Technologies Project Manager Workbench and marketed by ABT Technologies. In October 1992, Computer Power Group will release a version of Metrics Manager that integrates with Test/Cycle.

Future versions of Test/Cycle will incorporate Microsoft Word to support entry and editing of requirements. Additionally, user-defined reporting and reliability analysis will be supported.

27.4 Sample Outputs

Figures 27-1 through 27-22 provide sample outputs from Test/Cycle and Metrics Manager.

Page 1

Project: Gift Pack Order System
Wednesday, September 16, 1992
11:10:29

TEST/CYCLE REQUIREMENTS HIERARCHY REPORT

Ancestors:

None.

Parent:

Gift Pack Order System

Children:
---
1 System Entry (Main Window) +
2 Maintain Order File +
3 Order Inquiries +
4 Work Order Processing +
5 Order Pick-up/Delivery +
6 End-of-Day Processing +
7 Report Generation +

Page 2

TEST/CYCLE REQUIREMENTS HIERARCHY REPORT

Ancestors:

Gift Pack Order System

Parent:
---
1 System Entry (Main Window)

Children:

1.1 Menu Selection +
1.2 Maintain User IDs -
1.3 Maintain Statistics -
1.4 Maintain Parameter Values -

Page 3

TEST/CYCLE REQUIREMENTS HIERARCHY REPORT

Ancestors:

Figure 27-1. Test/Cycle Requirements Hierarchy Report

```
  Gift Pack Order System
1 System Entry (Main Window)

Parent:
-------
1.1 Menu Selection

Children:
---------
1.1.1  Enter GPOS System              -
1.1.2  Format Main Menu               -
1.1.3  Validate Menu Selection        -
1.1.4  Validate User ID               -
1.1.5  Xfer to Order Entry Scr        -
1.1.6  Xfer to Work Ords Scr          -
1.1.7  Xfer to Ord Pick-up Scr        -
1.1.8  Xfer to End of Day Proc Scr    -
1.1.9  Xfer to Ord Maint Scr          -
1.1.10 Exit System                    -
1.1.11 Disp "110 Invalid User ID" Msg -
1.1.12 Disp "120 Inv Select" Err Msg  -
1.1.13 Disp "Ord xxxxxx Entered" Msg  -
1.1.14 Xfer to Inquiry Scr            -
1.1.15 Xfer to Reports Scr            -
1.1.16 Xfer to Paran File Maint       -
                                    ...
                                                              Page 11

                TEST/CYCLE REQUIREMENTS HIERARCHY REPORT

Ancestors:
----------
  Gift Pack Order System
7 Report Generation

Parent:
-------
7.1 Report Menu

Children:
---------
1.2 Maintain User IDs             -
7.1.1 Generate Analysis Report    -
7.1.2 Print Work Orders on Request -
```

Figure 27-1 continued: Test/Cycle Requirements Hierarchy Report

Project: Gift Pack Order System
Wednesday, September 16, 1992
11:05:36

TEST/CYCLE REQUIREMENT DESCRIPTION REPORT

Requirement ID: 1.1 Menu Selection
Author: Jackie Jones

Requestor:
Testing Not Required: Not Applicable

Narrative:

No Narrative Found

Linkages:

To Characteristics:
None.

To Builds:
None.

To Test Cases:
None.

To Test Runs:
Main Menu Entries Validation

To Components:
None.

To Test Plans:
None.

To Work Requests:
None.

Figure 27-2. Test/Cycle Requirement Description Report

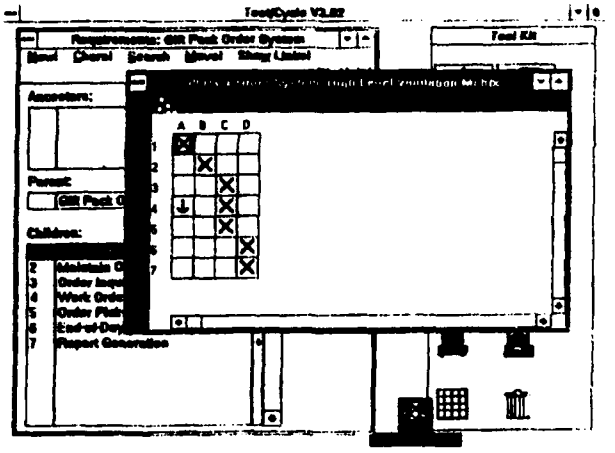


Figure 27-3. Test/Cycle High-Level Validation Matrix Screen

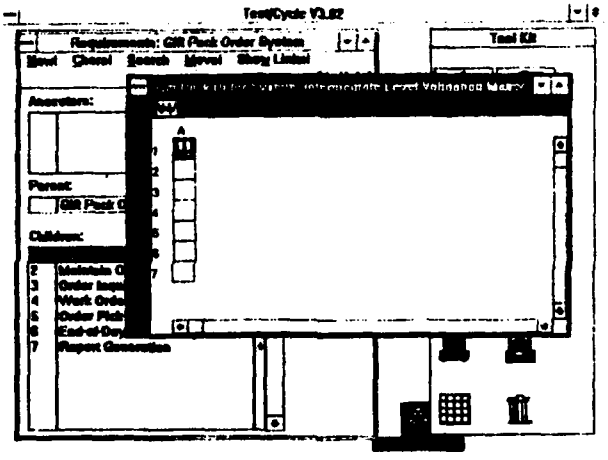


Figure 27-4. Test/Cycle Intermediate Level Matrix Screen

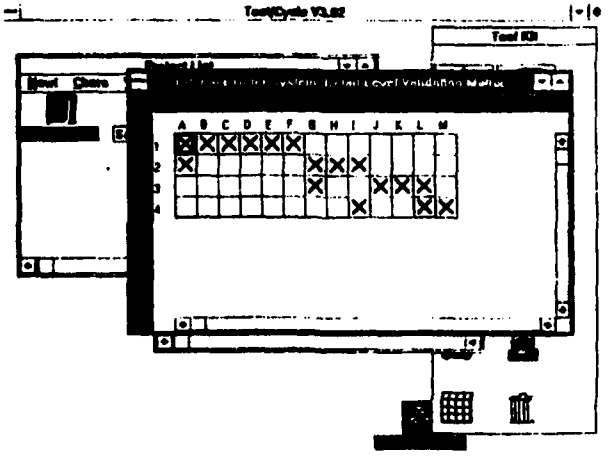


Figure 27-5. Test/Cycle Detail Level Matrix Screen

Project: Gift Pack Order System
Wednesday, September 16, 1992
11:24:27

TEST/CYCLE BUILD DESCRIPTION REPORT

Build ID: 1-System Entry & Param Maint
Author: Jackie Jones

Narrative:

This build consists of the main menu and parameter file maintenance functions. These functions allow the user to customize the system. The main menu function allows the user to select one of nine functions: Order Entry, Work Order Processing, Order Pick-Up, End-of-Day Processing, Order Maintenance, Inquiry, Report Generation, Parameter File Maintenance, or System Exit, When the user selects the exit function

Build Group Number:
Sequence Number:

Test Level Achieved: Integration Demoted: No

Linkages:

To Test Runs:
None.

To Requirements:
System Entry (Main Window) Update Ord Status to Complete

To Components:
None.

To Test Plans:
None.

To Work Requests:
None.

Figure 27-6. Test/Cycle Build Description Report

Project: Gift Pack Order System
Wednesday, September 16, 1992
11:31:44

TEST/CYCLE COMPONENTS DESCRIPTION REPORT

Component ID: Program1
Author: Jackie Jones

Narrative:

This is a sample of a program description. It is used to ...

Kind of Component: Program

Linkages:

To Requirements:
Order Enquiries

To Builds:
None.

To Test Cases:
None.

To Test Files:
None.

To Test Plans:
None.

To Work Requests:
None.

Figure 27-7. Test/Cycle Components Description Report

Project: Gift Pack Order System
Wednesday, September 16, 1992
11:28:22

TEST/CYCLE TEST RUN DESCRIPTION REPORT

Test Run ID: MM Select Validation
Author: Jackie Jones

Narrative:

The Main Menu Select Validation Test Run consists of a set of test cases that attempt to "break" the selection criteria of the main menu. In addition to verifying that the standard selection methods work, the test cases, which are a capture/playback of screen images, try to ...

Test Run Group Number:
Test Run Sequence:

Test Log:
Tom Smith 7/25/91

Linkages:

To Builds:
None.

To Requirements:
None.

To Test Cases:
None.

To Test Files:
None.

To Test Plans:
None.

To Work Requests:
None.

Figure 27-8. Test/Cycle Test Run Description Report

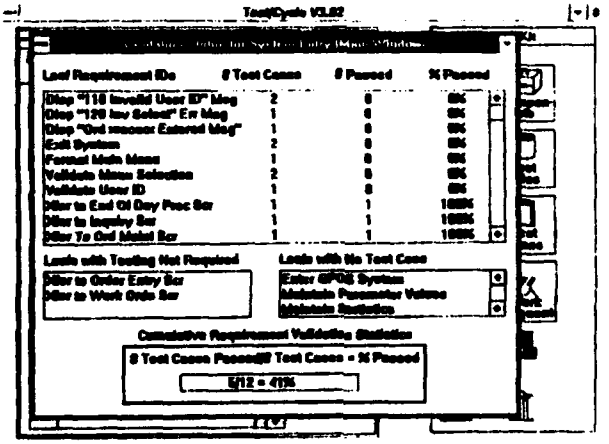


Figure 27-9. Test/Cycle Requirements Validation Status Screen

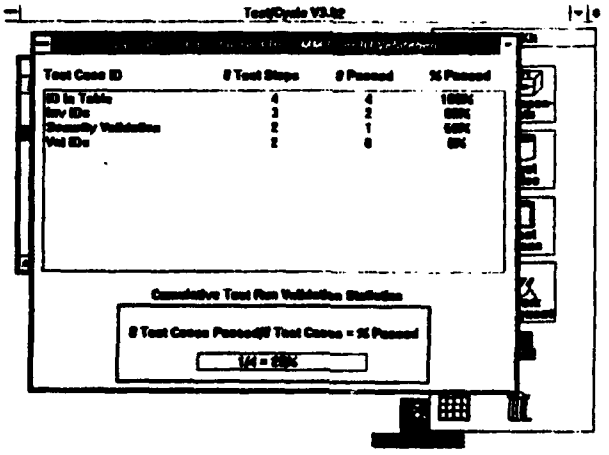


Figure 27-10. Test/Cycle Test Run Validation Status Screen

Page :

Project: Gift Pack Order System
Wednesday, September 16, 1992
11:40:18

TEST/CYCLE TEST CASE DESCRIPTION REPORT

Test Case ID: Inv IDs
Author: Jackie Jones
Tester:

Setup for Test:

System Started, Main Menu on Screen, cursor on User ID field

Test Steps:

	Pass	Description of Test	Expected Results	Action if Fail
1.	Yes	Hit <ENTER>	-Display ErrMsg "110 Invalid User ID" -Cursor on User ID. -User ID Highlighted	
2.	Yes	- Key "123" in User ID. Hit <ENTER>	-Display ErrMsg "110 Invalid User ID." -User ID Highlighted	

Linkages:

To Characteristics:
None.

To Requirements:

Disp "110 Invalid User ID" Msg	Format Main Menu
Display Ord Inf for Pick-up	Validate Menu Selection

Narrative:

The Parameter File must contain one or more valid user IDs for validation of the Main Menu User ID and Selection entries.

Figure 27-11. Test/Cycle Test Case Description Report

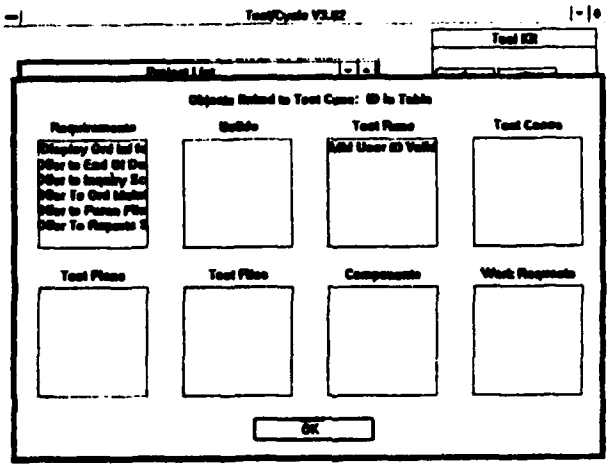


Figure 27-12. Test/Cycle Test Case Linkages Screen

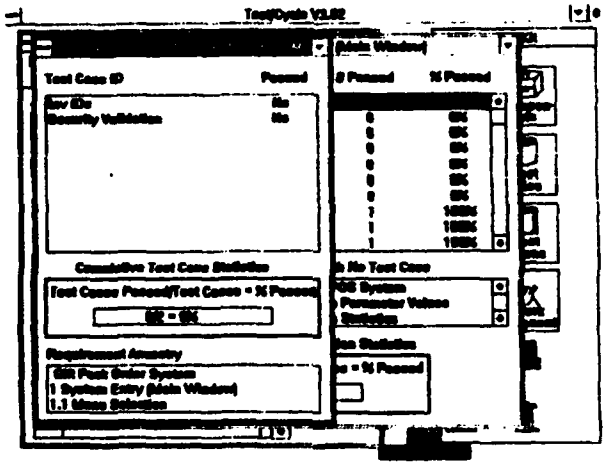


Figure 27-13. Test/Cycle Test Case Referenced by Requirement Screen

Project: Gift Pack Order System
Wednesday, September 16, 1992
11:36:03

TEST/CYCLE TEST FILE DESCRIPTION REPORT

Test File ID: Main Menu Val IDs Param File
Author: Jackie Jones

Narrative:

The Parameter File must contain one or more valid user IDs for validation of the Main Menu User ID and Selection entries.

Test File Type: Sequential

Linkages:

To Test Runs:
None.

To Test Cases:
None.

To Components:
None.

To Work Requests:
None.

Figure 27-14. Test/Cycle Test File Description Report

```
ID: 1                                          Category: Open Problem
Instigator: Jackie Jones
Requestor:
WR Date:    1/8/92                             Priority:

WR Type:

Narrative:
----------
This is a sample work request for a problem report. It is used to ...

Status: None

Error: No Problem metrics record found for 1

Linkages:
---------
Paradox ISAM Emulator(Cmp).
```

Figure 27-15. Test/Cycle Work Request Description

```
                                                                    Page 1
Project:  Gift Pack Order System
Wednesday, September 16, 1992
11:44:10

                TEST/CYCLE WORK REQUEST LOG REPORT

Open Problem Work Requests
--------------------------

Cat  ID    Priority   Type          Narrative                  Status
---  ----- -------- -------- ------------------------------ --------
OP       1                   <some text>                    None
OP       2                   <some text>                    None
```

Figure 27-16. Test/Cycle Work Request Log Report

SAMPLE OUTPUTS FROM MARS

```
09/24/92  11:20                  METRICS MANAGER                    Page   1
[MDB001] V3.02                   METRICS DATA BASE
                                 Enterprise: 5550

                              PERIOD ENDING SEP, 1992


ENTERPRISE
----------

TOTAL SALES REVENUE:                    $700,000,000

TOTAL NUMBER OF EMPLOYEES:                 6,000

SIC CODE:                                     -1
SECONDARY SIC CODE:                           -1
```

Figure 27-17. Metrics Manager Database Full Report

09/24/92 11:20 METRICS MANAGER Page 2
[MDB002] V3.02 METRICS DATA BASE
Enterprise: 5550
MIS: Management Information Systems

PERIOD ENDING SEP, 1992

MIS

HARDWARE BUDGET:	$2,000,000
SOFTWARE BUDGET:	$500,000
PERSONNEL BUDGET:	$7,000,000
FACILITIES BUDGET:	$2,000,000
OTHER BUDGET:	$2,000,000
TOTAL BUDGET:	$13,500,000

TOTAL FUNCTION POINTS OF INSTALLED BASE:	0
NUMBER OF TERMINALS USED FOR DEVELOPMENT AND MAINTENANCE --	
Mainframe Terminals:	150
PC Workstations:	100
TOTAL MIPS:	46.1
TOTAL PROGRAMS:	4,582
TOTAL KLOC:	2,726
TOTAL STAFF:	315
Total Maintenance Staff:	-1
Total Technical Staff:	85
AVERAGE DEPARTMENT LABOR RATE:	$25.00
MAINTENANCE LABOR RATE:	$-1.00

Figure 27-17 continued: Metrics Manager Database Full Report

09/24/92 11:20 METRICS MANAGER Page 3
[MDB003] V3.02 METRICS DATA BASE
Enterprise: 5550
MIS: Management Information Systems
Product: Product A - Development
PERIOD ENDING SEP, 1992

PRODUCT ATTRIBUTES

TYPE OF EFFORT:	A - New Development
DEVELOPMENT END DATE:	04/88
DATE OF ORIGINAL REQUEST:	/ /
DATE NEEDED:	02/27/87
DATE STARTED:	05/31/86
DATE OF FIRST PRODUCTION UTILIZATION:	04/30/88
DATE LAST COMPONENT INSTALLED:	04/30/88
DURATION:	700 DAYS
BACKLOG PERIOD:	0 DAYS

TARGET DATES
Original: 01/01/88 Revised: / / Actual: 01/01/88

DAYS VARIANCE:	0 DAYS
TOTAL KLOC:	29

FUNCTION POINT COUNTS (Actual Approved)

Adjusted Function Points (IFPUG):	1,630
Function Points Override:	0

USER INVOLVEMENT

Definition Phase:	H
Construction Phase:	H
Operation Phase:	H

PERCENT OF MIS TO TOTAL PERSONNEL

Definition Phase:	90%
Construction Phase:	90%
Operation Phase:	95%

PROJECT MANAGER'S EXPERIENCE:	2 Years
STAFF APPLICATION EXPERIENCE:	2 Years
TECHNICAL YEARS OF EXPERIENCE:	6 Years
SYSTEM AVAILABILITY:	95%
SYSTEM RESPONSE TIME:	5 Seconds
NET PRESENT VALUE:	-1
RETURN ON INVESTMENT:	-1.00%
ACTUAL PEAK TEAM SIZE:	5

Figure 27-17 continued: Metrics Manager Database Full Report

09/24/92 11:20 METRICS MANAGER Page 4
[MDB004] V3.02 METRICS DATA BASE
Enterprise: 5550
MIS: Management Information Systems
Product: Product A - Development

PERIOD ENDING SEP, 1992

PRODUCT COST

DEVELOPMENT COSTS	Original	Revised	Actual
Labor	$350,868	$575,984	$737,523
Hardware	$70,000	$120,296	$263,153
Expense	$0	$0	$0
TOTAL	$420,868	$696,280	$1,000,676

ACTUAL TECHNICAL LABOR COST: $0

TOTAL PRODUCTION COST: $10,400
Production Rerun Cost: $-1

DEFECT REMOVAL COSTS	Definition Phase	Construction Phase	Operation Phase	TOTAL
Labor	$2,600	$14,325	$33,575	$50,500
Machine	$-1	$5,900	$14,800	$20,700
TOTAL	$2,600	$20,225	$48,375	$71,200

FAILURE COSTS
Internal Labor: $25,900
Internal Machine: $1,000
Total Internal Failure: $26,900
External Failure: $-1
Production Rerun Cost: $-1
TOTAL FAILURE COST: $26,900

Figure 27-17 continued: Metrics Manager Database Full Report

09/24/92 11:20 METRICS MANAGER Page 5
[MDB005] V3.02 METRICS DATA BASE
Enterprise: 5550
MIS: Management Information Systems
Product: Product A - Development

PERIOD ENDING SEP, 1992

PRODUCT EFFORT (HOURS)

DEVELOPMENT LABOR EFFORT	
Original:	8,354 HOURS
Revised:	12,058 HOURS
Actual:	15,365 HOURS
ACTUAL TECHNICAL	
LABOR EFFORT:	0 HOURS
DEFECT REMOVAL EFFORT	
Definition Phase:	104 HOURS
Construction Phase:	573 HOURS
Operation Phase:	1,343 HOURS
TOTAL:	2,020 HOURS
INTERNAL FAILURE EFFORT:	1,036 HOURS

Figure 27-17 continued: Metrics Manager Database Full Report

09/24/92 11:20 METRICS MANAGER Page 6
[MDB006] V3.02 METRICS DATA BASE
Enterprise: 5550
MIS: Management Information Systems
Product: Product A - Development

PERIOD ENDING SEP, 1992

PRODUCT QUALITY

DEFECTS BY INSERTION PHASE	Minor	Moderate	Severe	Total
In Definition	-1	85	-1	85
In Construction	-1	66	-1	66
In Operation	-1	167	-1	167
TOTAL	0	318	0	318

DEFECTS BY DETECTION PHASE	Minor	Moderate	Severe	Total
Definition	-1	26	-1	26
Construction	-1	125	-1	125
Operation	-1	167	-1	167
TOTAL	0	318	0	318

	Minor	Moderate	Severe	Total
FAILURES	-1	126	-1	126

Figure 27-17 continued: Metrics Manager Database Full Report

09/24/92 11:20 METRICS MANAGER Page 7
[MDB007] V3.02 METRICS DATA BASE
Enterprise: 5550
MIS: Management Information Systems
Product: Product A - Development
PERIOD ENDING SEP, 1992

PRODUCT METRICS

MONTHS IN OPERATION:	12 Months
PRODUCTIVITY	
KLOC Per Staff Month:	0.327
Function Points Per Staff Month:	18.388
CYCLE TIME (Elapsed Days / Function Point)	
Overall Cycle Time:	0.000 Days
Development Cycle Time:	0.429 Days
COST / FUNCTION POINT	
Development Unit Cost:	$613.91
Development Labor Unit Cost:	$452.40
Development Defect Cost Rate:	$14.00
COST PER MONTH / FUNCTION POINT	
Production Defect Cost Rate:	$2.47
Failure Cost Rate:	$1.38
Production Unit Cost Rate:	$0.53
RELIABILITY	
Mean Time To Failure:	0.0952 Months per Failure
Monthly Failure Rate:	0.0064 Failures / Month / FP
Monthly Failure Rate 1st 6 Months:	0.0090 Failures / Month / FP
Monthly Failure Rate Last 6 Months:	0.0039 Failures / Month / FP
Failures Per Execution Hour:	9.0000 Failures per Hr
Failure Rate Per Execution Hour:	0.0055 Failures per Hr / FP
PERFORMANCE	
Effort Variance:	27.00%
Cost Variance:	44.00%
Schedule Variance:	0.00%
QUALITY	
Defect Density:	0.195 Defects / FP
Development Defect Removal Efficiency (Development Defects / All Defects):	0.475

CUSTOMER SURVEY METRICS:

	User Satisfaction With SDP	Strategic Value	Tactical Value
8804:	N/A		

Figure 27-17 continued: Metrics Manager Database Full Report

```
09/24/92  11:20                  METRICS MANAGER                    Page    8
[MDB003] V3.02                   METRICS DATA BASE
                                 Enterprise: 5550
                      MIS: Management Information Systems
                    Product: Product B - 1st Enhancement

                           PERIOD ENDING SEP, 1992

PRODUCT ATTRIBUTES
------------------

TYPE OF EFFORT:          B - Major Enhancement

DEVELOPMENT END DATE:                                01/89

DATE OF ORIGINAL REQUEST:                         09/01/87
DATE NEEDED:                                      01/01/89
DATE STARTED:                                     06/01/87
DATE OF FIRST PRODUCTION UTILIZATION:             01/01/89
DATE LAST COMPONENT INSTALLED:                    01/31/89
DURATION:                                         610 DAYS
BACKLOG PERIOD:                                   -92 DAYS

TARGET DATES
  Original:  10/01/88     Revised:     / /       Actual:  01/31/89

DAYS VARIANCE:                                    122 DAYS

TOTAL KLOC:                                             1
                                  ...
```

Figure 27-17 continued: Metrics Manager Database Full Report

09/25/92 09:52 METRICS MANAGER Page 1
[MDB008] V3.02 METRICS DATA BASE
Enterprise: 5550
MIS: Management Information Systems

PERIOD ENDING SEP, 1992

MIS SUMMARY

Total products in MIS: 5

Basic Measures		Metrics	
		KLOC PER STAFF MONTH:	1.027
TOTAL KLOC:	314	F/P PER STAFF MONTH:	18.388
TOTAL FUNCTION POINTS:	1,630	Cost / Function Point	
		DEVELOPMENT UNIT COST:	$613.91
ACTUAL LABOR EFFORT:	52,981	DEVEL. LABOR UNIT COST:	$452.47
ACTUAL DEVELOPMENT COST:	$2,874,715	DEV. DEFECT COST RATE:	$14.00
TOTAL DEFECT REMOVAL COST:	$82,500	PROD. DEFECT COST RATE:	$2.47
TOTAL FAILURE COST:	$32,075	FAILURE COST RATE:	$1.38
		PROD. UNIT COST RATE:	$0.53
TOTAL DEFINITION DEFECTS:	50	Quality	
TOTAL CONSTRUCTION DEFECTS:	152		
TOTAL OPERATION DEFECTS:	175	DEFECT DENSITY PER F/P:	0.1951
TOTAL DEFECTS:	377	DEVEL. DEFECT REMOVAL EFFICIENCY:	0.536
TOTAL DEFECTS INSERTED IN --		Reliability	
DEFINITION:	111		
CONSTRUCTION:	91	FAILURES PER EXECUTION HOUR:	2.3276
OPERATION:	175		
TOTAL FAILURES:	135		

Figure 27-18. Metrics Manager Enterprise & MIS Metric Summary Report

09/25/92 09:52 METRICS MANAGER Page 2
[MDB008] V3.02 METRICS DATA BASE
Enterprise: 5550

PERIOD ENDING SEP, 1992

ENTERPRISE SUMMARY

Total products in ENTERPRISE: 5

Basic Measures		Metrics	
		KLOC PER STAFF MONTH:	1.027
TOTAL KLOC:	314	F/P PER STAFF MONTH:	18.388
TOTAL FUNCTION POINTS:	1,630	Cost / Function Point	
		DEVELOPMENT UNIT COST:	$613.91
ACTUAL LABOR EFFORT:	52,981	DEVEL. LABOR UNIT COST:	$452.47
ACTUAL DEVELOPMENT COST:	$2,874,715	DEV. DEFECT COST RATE:	$14.00
TOTAL DEFECT REMOVAL COST:	$82,500	PROD. DEFECT COST RATE:	$2.47
TOTAL FAILURE COST:	$32,075	FAILURE COST RATE:	$1.38
		PROD. UNIT COST RATE:	$0.53
TOTAL DEFINITION DEFECTS:	50	Quality	
TOTAL CONSTRUCTION DEFECTS:	152		
TOTAL OPERATION DEFECTS:	175	DEFECT DENSITY PER F/P:	0.1951
TOTAL DEFECTS:	377	DEVEL. DEFECT REMOVAL EFFICIENCY:	0.536
TOTAL DEFECTS INSERTED IN --		Reliability	
DEFINITION:	111		
CONSTRUCTION:	91	FAILURES PER EXECUTION HOUR:	2.3276
OPERATION:	175		
TOTAL FAILURES:	135		

Figure 27-18 continued: Metrics Manager Enterprise & MIS Metric Summary Report

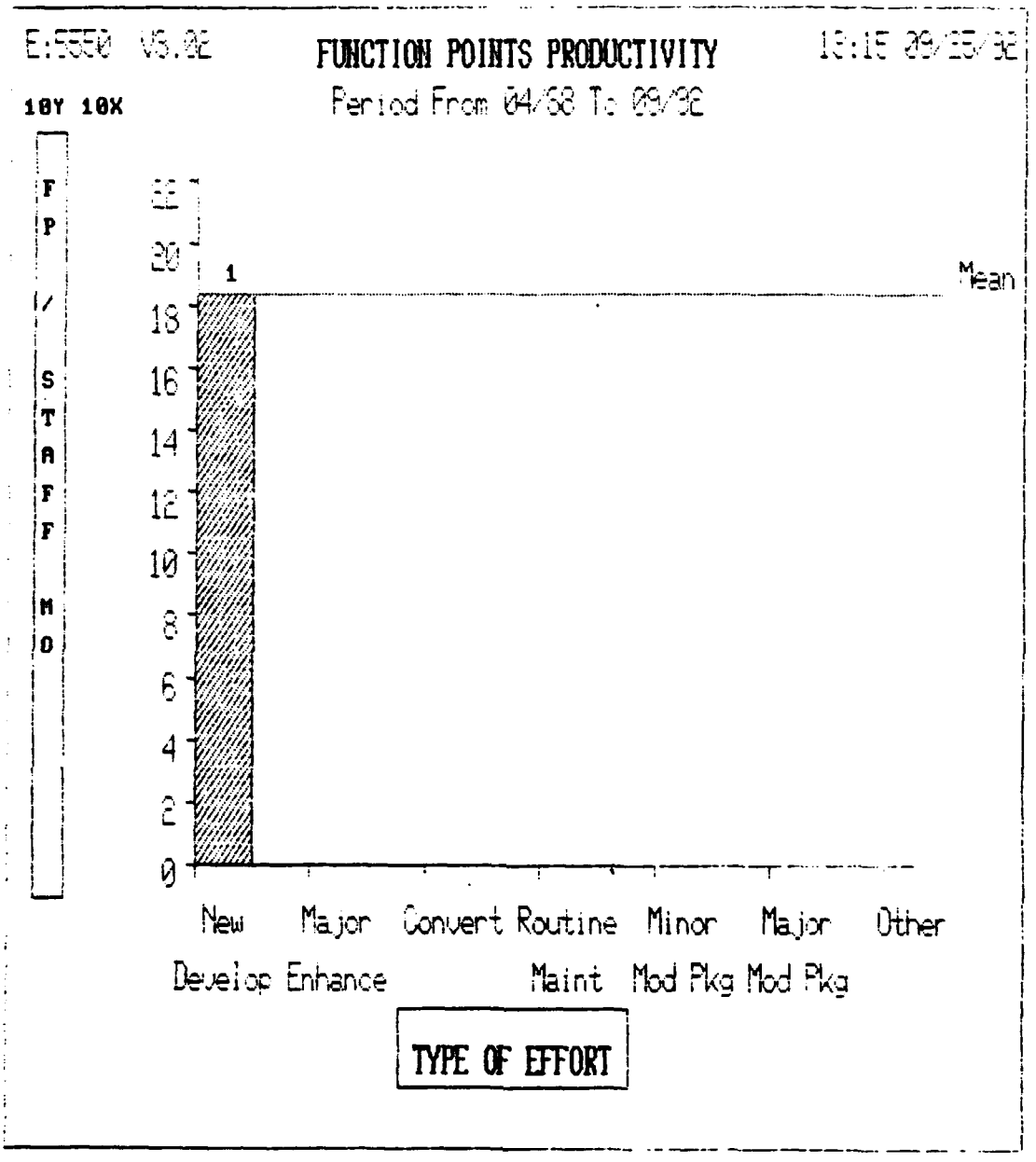


Figure 27-19. Metrics Manager Function Points Productivity vs. Type of Effort

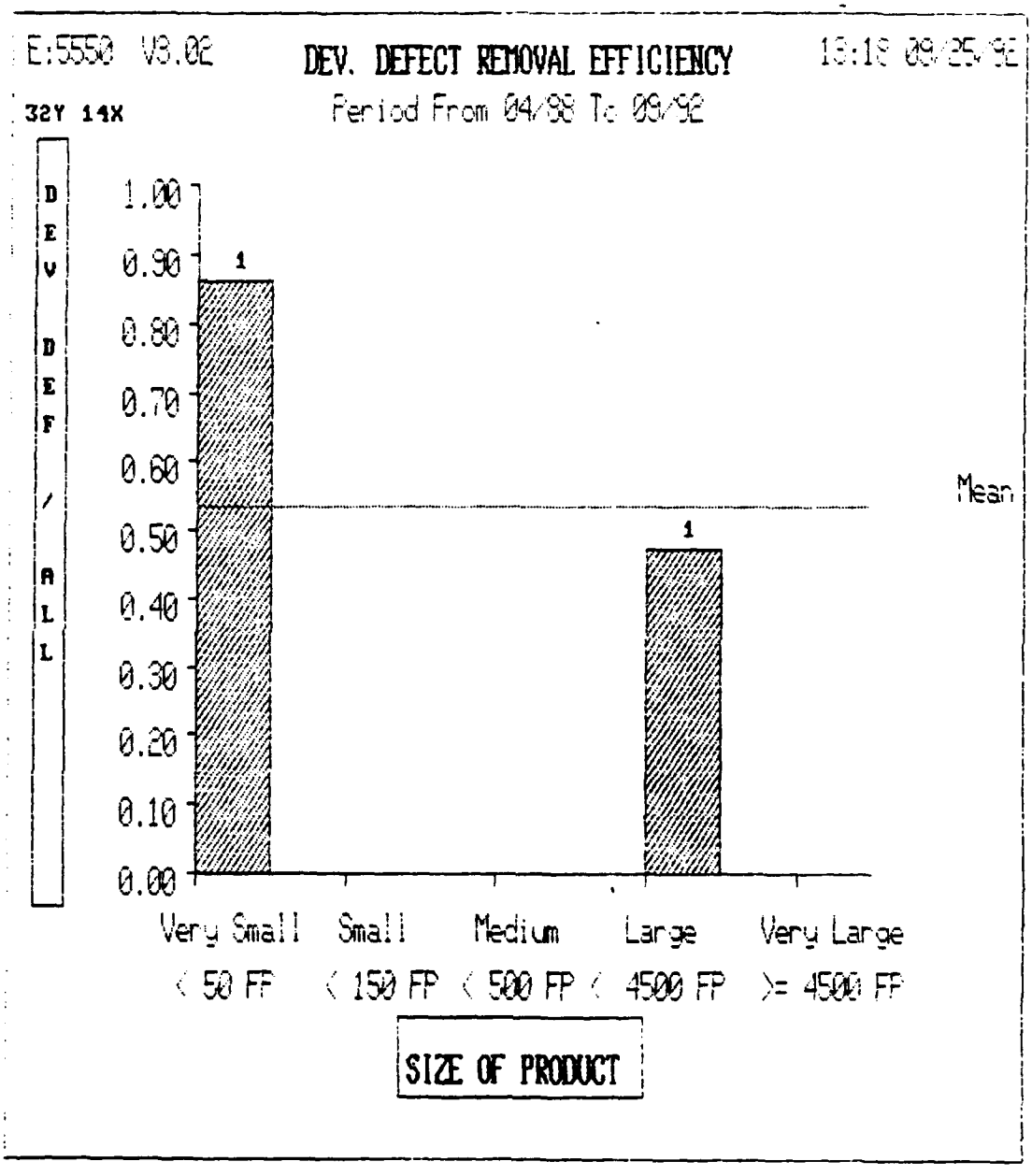


Figure 27-20. Metrics Manager Development Defect Removal Efficiency vs. Size of Product

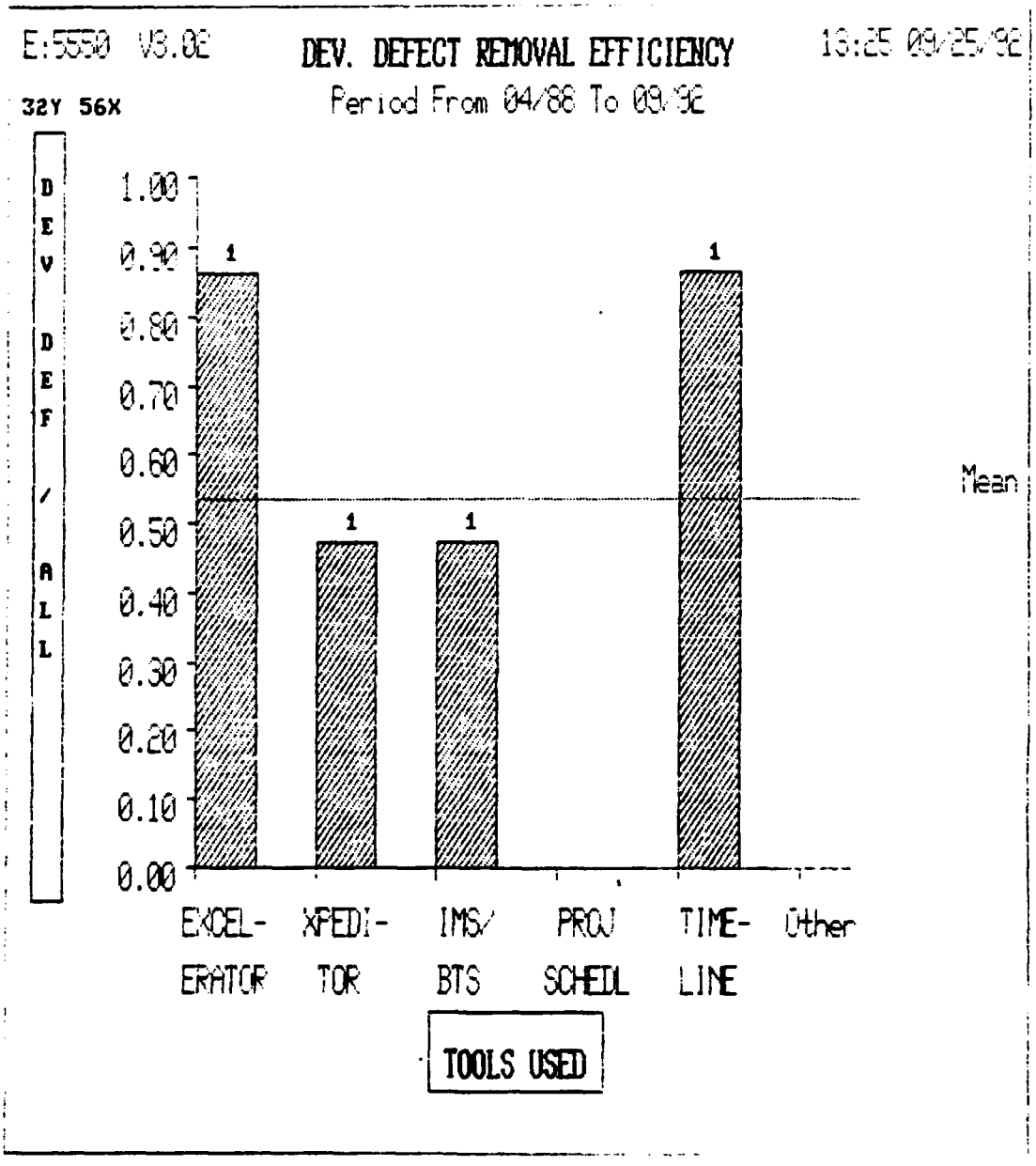


Figure 27-21. Metrics Manager Development Defect Removal Efficiency vs.Tools Used

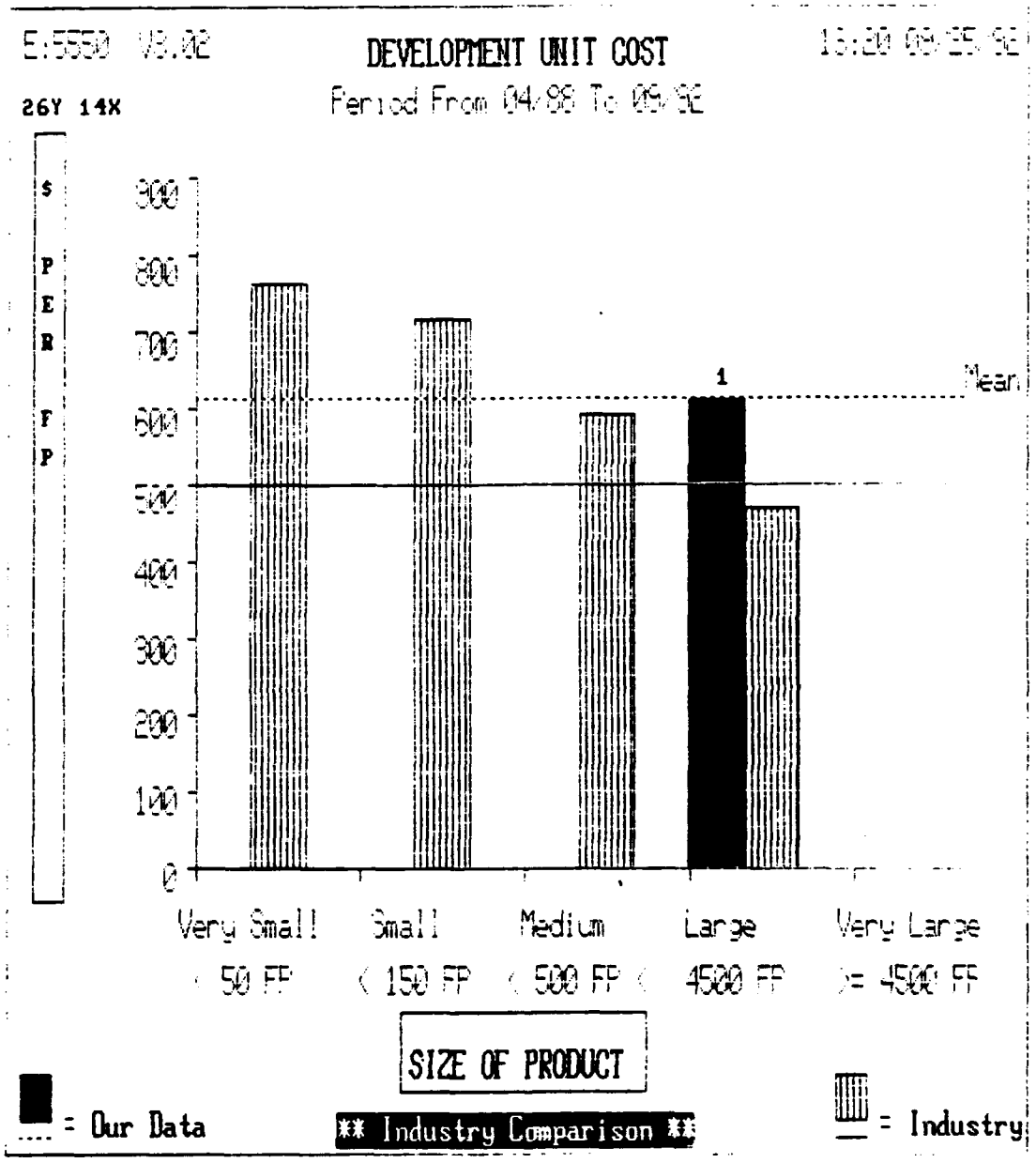


Figure 27-22. Metrics Manager Development Unit Cost vs. Size of Product Showing Industry Data

28. TestGen, QualGen, GrafBrowse, and the ADADL Processor

TestGen, QualGen, GrafBrowse, and the ADADL processor are part of the AISLE family of software tools based on the Ada-based Design and Documentation Language (ADADL). ADADL itself is fully compilable by any Ada compiler and has been selected by the Joint Integrated Avionics Working Group (JIAWG) as the Ada program design language (PDL) to use for the Army's Light Helicopter (LH) and the Air Force's Advanced Tactical Fighter (ATF) programs.

The ADADL processor provides static analysis of ADADL designs and Ada code, and produces the outputs needed by TestGen, QualGen, and GrafBrowse. These other tools also operate on both ADADL designs and Ada code. TestGen supports the review of ADADL designs, preparation of unit test plans, and test coverage analysis. QualGen reports on quality, using a user-tailorable metrics hierarchy. GrafBrowse supports code browsing and creates invocation trees to assist in reverse engineering.

Other AISLE tools include syntax-directed and graphical editors, an automatic document generator, a design database analyzer, a requirements analyzer and tracer, and a compilation order analyzer.

28.1 Tool Overview

The AISLE tool family is marketed by Software Systems Design. It has been available since 1984 and has over 1,000 users. The tools are available on a wide range of machines such as VAX, VAXStation, and MicroVAX under VMS, Unix, or Ultrix; and Sun-3 and Sun-4, HP9000-800, Apollo, DecStation, and 80386-based PC systems under MS-DOS or Unix. Where windowing is required, the tools support X-Windows, OpenWindows, SunWindows, DECwindows, Tektronix, and Hewlett-Packard windows. Graphics output is formatted for a range of devices. These formats include Postscript, Tektronix, and Graphical Kernel System (GKS). AISLE can interface to the Teamwork, StP, and Excelerator CASE systems to provide automatic generation of designs from requirements. At the time of examination, TestGen prices started at $4,600, the ADADL processor at $5,000, QualGen at $4,000, and GrafBrowse at $5,500. Training and consulting services are available.

The IDA study used the ADADL processor version 5.3.E, TestGen version 2.2.2, QualGen version 1.1, and GrafBrowse version 2.2.2 running on a Sun-4 system under Open-

Windows. The examinations focused on application of the tools to Ada code rather than to ADADL designs.

28.1.1 ADADL Processor Overview

The user starts the application of these tools by submitting the code to the ADADL processor. When the code exists in several files, these must be submitted in compilation order. The processor itself consists of over 25 tools, although these are transparent to the user. While some of these tools operate only on ADADL designs, the majority apply to either ADADL designs and Ada code, or just to Ada code. This latter category of tools includes the following:

- A pretty printer.
- An object highlighter to highlight use of Ada entities.
- A program unit invocation tree generator.
- A program unit declaration hierarchy generator.
- Several cross referencers, including an object, type, subtype and derived type parent reference cross referencers.
- Generic instantiation report shows the location of each generic instantiation.
- Interrupt report generator shows how interrupts are declared and where they are used.
- *With* hierarchy generator shows the library units imported by each program unit.
- Quality analyzer identifies design or code portions that do not conform to quality guidelines, for example, objects declared but not used, and program units with an ADADL complexity greater than some maximum limit.
- Complexity analyzer computes the cyclomatic complexity and ADADL complexity measures.
- Undefined identifier/spelling checker identifies possible spelling errors or the omission of a declaration.

Report formatting can be modified, or the production of specific reports disabled, using ADADL processor commands. Like ADADL design statements, these are given in the form of special Ada comments. Additional special comments include the following:

- Data dictionary definitions. Used to produce a data dictionary of all Ada program units, types, and objects.
- Project management information. Used to define information useful to the project manager, for example, dates of completion of design, coding, and testing activities.

The use of these special comments was not examined.

28.1.2 TestGen Overview

TestGen supports structural testing at three levels: branch testing, structured testing based on McCabe's cyclomatic complexity number (also called basis path testing) [McCabe 1976], and path testing. Using a selected technique, TestGen gives the number of necessary test cases and then, for each test case, identifies the program conditions required at each decision point to exercise the necessary program paths and shows the statements that will be executed during that test. This information helps the user derive necessary test data for structural testing. Since each level of testing may require a potentially very large number of test cases, TestGen provides an option that allows the user to first see how many test cases will be required for each program unit using the different testing techniques. This information can be used to guide subsequent test case generation; it is also useful in estimating testing costs in terms of the number of test cases required for structural analysis.

Before instrumenting code for coverage data collection, the user must establish a TestGen library. Special utilities are provided for such library creation and maintenance. Once an appropriate library has been created and opened, the user specifies the files that should be instrumented. The user can further limit the extent of instrumentation by requesting instrumentation of selected program units instead of all the units in the file. The instrumentation process also requires the creation of a simple test driver. If the program under test contains a main procedure, TestGen can automatically generate the necessary driver. (This driver performs a loop calling the instrumented program for as long as the user wishes.) Otherwise, a special test driver must be manually created by the user based on a template supplied in the documentation.

The user manually compiles and links the instrumented code and test driver. When invoked, the test driver queries the user for a run identification and brief description. As the program executes on test data, the instrumentation produces a trace history that records the order in which statements were executed. (The run identification is used in naming the generated trace file.) The Test Coverage Analyzer is then used to analyze this trace history and report on the coverage achieved. For each program unit, a Test Coverage Summary report identifies the number of times that unit was executed, a count of the statements not executed, and the number of branch, basis, and complete paths that were executed. Further details on which statements were executed and how many times these were exercised are provided by annotating a program listing. Similarly, the user can request a report that details those paths that were not exercised. Reporting on the coverage accumulated over a series of test runs is achieved by requesting analysis of multiple trace histories. Although this reports on

the total coverage achieved with the related test data, the Test Coverage Summary does not distinguish between the coverage achieved on different runs.

Additional TestGen utilities are provided to compute the cyclomatic complexity for each selected program unit, identify any unexecutable paths, and provide a control graph of the code. A final utility, the Design Review Expert Assistant, was not examined.

28.1.3 QualGen Overview

QualGen is currently packaged as part of the ADADL processor. It provides analysis of both design and code metrics. QualGen comes with more than two hundred primitive metrics divided into major categories such as complexity, modularity, documentation, error handling, system independence, and clean up. These metrics are taken from the works of quality experts such as Boehm, Halstead, and McCabe. The Software Productivity Consortium style guide was another a source of quality measurement guidance used by Software Systems Design. The user can select which of the primitive metrics will be reported on.

QualGen results are imported into Lotus 1-2-3 for further analysis and reporting. Using Lotus, the user can create formulae that define how primitive metrics should be combined to yield higher-level metrics such as completeness, reliability, and portability. Lotus also supports the preparation of graphical presentation of quality results.

28.1.4 GrafBrowse Overview

Primarily intended to support reverse engineering, GrafBrowse facilitates program understanding by allowing the user to graphically view the interrelationships between Ada entities. As with the ADADL processor, it operates on both an ADADL design and Ada code. Again, however, the IDA examination focused on the application of GrafBrowse to Ada code.

When invoked, GrafBrowse presents the program invocation tree. Alternatively the user can select a declaration tree or called-by tree. (The invocation and called-by trees can be displayed in compact form, that is, with lines that potentially cross, or in a flat view without crossing lines.) Where appropriate, the user can follow this selection with another to select particular program units to focus on. Once a picture is displayed, a pop-up menu allows the selection of a new view, browsing the related source code, or presentation of any definition provided for the unit. Another menu allows some reformatting of the displayed

view, annotating each unit representation with its formal parameters, and printing the view. Since many of the generated charts will be large, the print option allows the user to request reducing a view to fit on a page, dissecting and scaling a chart for presentation on connected sheets of paper, or printing a chart as it appears on the screen using as many sheets of paper as necessary.

28.2 Observations

Ease of use. The tools may be invoked independently from the command line. Alternatively, the AISLE user interface provides a graphical interface to their use. In either event, all the tools are menu driven. An on-line manual is available to provide on-line help. Error messages are terse and only minimal checking of user input is provided. No special knowledge is required to use the tools. All output reports are well-structured and provide easy-to-read information. A major strength of this tool is the clear identification of the path conditions that guide the execution of particular program paths. Other than setting default values for files names, the user interface cannot be tailored.

Documentation and user support. The documentation was helpful and included several useful examples. Software Systems Design staff provided quick and helpful support.

Instrumentation overhead. The entire program must be analyzed by the ADADL processor before the Unit Test Strategy Generator can be used. Subsequently, TestGen functions can be invoked for the files analyzed by the ADADL processor. The size of instrumented code is minimized by allowing the user to specify which modules in the selected file should be instrumented. All selected modules are instrumented in the same fashion. Instrumentation of the Ada Lexical Analyzer Generator gave a 150% increase in code size, and an increase of 27% in the object code.

Ada restrictions. The Unit Test Strategy Generator can analyze any Ada code, but the Test Coverage Analyzer cannot instrument tasks. Source lines with multiple statements may be instrumented incorrectly.

Problems encountered. Execution of the fully instrumented Ada Lexical Generator caused storage errors to be raised. It is uncertain whether this problem was due to the amount of tracing data being generated or to a fault in the instrumentation itself. Software Systems Design are investigating this problem.

28.3 Planned Additions

Software Systems Design is currently revising QualGen. This tool is being extracted from the ADADL processor and being made independent of Lotus 1-2-3. It will itself generate histograms and Kiviat diagrams for presentation of metrics data and allow the user to combine metrics into composite quality measures. The new version will also provide for trend analysis of metrics data.

A new tool, called BugFinder, will examine program paths to look for potentially erroneous conditions, for example, a path in which the output is not set. This tool is expected to become available in April 1993.

28.4 Sample Outputs

Figures 28-1 through 28-29 provide sample outputs from the ADADL processor, TestGen, QualGen, and GrafBrowse.

```
[][][][][][][][][][][][][][][][][][][][][][][][][][][][]
[][][][]                                        [][][][]
[][][][]                                        [][][][]
  [][]            ADADL PROCESSOR                 [][]
  [][]                                            [][]
  [][]             VERSION 5.3.E                  [][]
  [][]         SERIAL NUMBER: LOAN-IDA            [][]
  [][]                                            [][]
  [][]            AUTHORIZED USER                 [][]
  [][]          Software Systems Design           [][]
  [][]                                            [][]
  [][]                                            [][]
[][][][]                                        [][][][]
[][][][][][][][][][][][][][][][][][][][][][][][][][][][]
                          ...
```

```
LINE                                                              PAGE 4
   function  Llfind( Item: Llstrings; Which: Llstyle )  return  Integer &
      is
     --  Find item in symbol table -- return index or 0 if not found.
     --  Assumes symbol table is sorted in ascending order.
     Low, Midpoint, High: Integer;
   begin
                            ********************
                            *                  *
                            * ADA CODE FOLLOWS *
                            *                  *
                            ********************

     Low := 1;
     High := Lltablesize + 1;
     while  Low /= High  loop
     |  Midpoint := (High + Low) / 2;
     |  if  Item < Llsymboltable(Midpoint).Key  then
     |  |  High := Midpoint;
     |  elsif  Item = Llsymboltable(Midpoint).Key  then
     |  |  if  Llsymboltable(Midpoint).Kind = Which  then
<--------------return( Midpoint );
     |  |  else
<--------------return( 0 );
     |  |  end if;
     |  else   --   ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
     |  |  Low := Midpoint + 1;
     |  end if;
     end loop;
<-----return( 0 );  -- item is not in table
   end  Llfind;
```

Figure 28-1. ADADL Listing

```
                                                                    PAGE 42
                        PROGRAM UNIT CROSS REFERENCE REPORT
================================================================================
                    DEC/REF          LOCATION OF DECLARATION OR REFERENCE
                                   PAGE NO.    LINE NO.    ENCLOSING PROGRAM UNIT

Advance    <<Procedure specification>>
                    declaration      15          248      Package Ll_Tokens

Buildright    <<Procedure>>
                    declaration      19          302      Procedure Readgram
                    code ref         21          394

Buildselect    <<Procedure>>
                    declaration      21          358      Procedure Readgram
                    code ref         21          395

Close    <<Procedure specification>>
                    declaration     ** LIBRARY **         Package Text_Io
                    code ref         22          403      Procedure Readgram
                                     ...

Llfind    <<Function - Returns Integer>>
                    declaration       4           83      Procedure Ll_Compile
                    code ref         13          215      Function Make_Token
                    code ref         13          217
                    code ref         13          219
                    code ref         13          222
                    code ref         13          224
                    code ref         13          227
                    code ref         29          563      Procedure Parse
                    code ref         29          565

Llmain    <<Procedure>>
                    declaration      17          269      Procedure Ll_Compile
                    code ref         30          613

Llnexttoken    <<Procedure>>
                    declaration       3           79      Procedure Ll_Compile
                    code ref          7          136      Procedure Llskiptoken
                    code ref          9          163      Procedure Llskipboth
                    code ref         29          534      Procedure Synchronize
                    code ref         29          566      Procedure Parse
                    code ref         30          573
                    declaration      16          255      Procedure Ll_Compile
                                     ...
```

Figure 28-2. ADADL Program Unit Cross Reference Report

PAGE 47

OBJECT CROSS REFERENCE REPORT

	DEC/REF	LOCATION OF DECLARATION OR REFERENCE PAGE NO.	LINE NO.	ENCLOSING PROGRAM UNIT
Attribute <<Object>>				
declared as => Llattribute				
	declaration	12	196	Function Make_Token
set	code ref	13	233	
set	code ref	13	235	
set	code ref	14	237	
set	code ref	14	239	
set	code ref	14	241	
use	code ref	14	243	
Axiom <<Object>>				
declared as => Integer				
	declaration	17	292	Procedure Llmain
parameter	code ref	21	387	Procedure Readgram
use	code ref	29	561	Procedure Parse
Caseindex <<In Parameter>>				
declared as => in Integer				
	declaration	16	266	Procedure Lltakeaction
Ch <<Object>>				
declared as => Character				
	declaration	18	298	Procedure Readgram
parameter	code ref	19	311	Procedure Buildright
use	code ref	19	312	
parameter	code ref	21	377	Procedure Readgram
use	code ref	21	379	
Childcount <<Object>>				
declared as => Integer				
	declaration	19	303	Procedure Buildright
set	code ref	19	307	
set	code ref	19	314	
use	code ref	19	314	
use	code ref	19	315	
set	code ref	19	321	
use	code ref	19	321	
use	code ref	19	322	
set	code ref	19	326	
use	code ref	19	326	
use	code ref	19	327	
Cr <<Object>>				
declared as => constant Character := CR				
	declaration	** LIBRARY **		Package Ascii
use	code ref	11	185	Procedure & Get_Character
		...		

Figure 28-3. ADADL Object Cross Reference Report

```
                                                                    PAGE 58
                        TYPE CROSS REFERENCE REPORT
================================================================================
                    DEC/REF          LOCATION OF DECLARATION OR REFERENCE
                                   PAGE NO.    LINE NO.    ENCLOSING PROGRAM UNIT

Boolean   <<Type>>
     declared as =>  (False, True)
                    declaration     ** LIBRARY **          Package Standard
                    code dec          3            60     Procedure Ll_Compile
                    code dec          3            68
                    code dec          3            69
                    code dec         11           178     Procedure  &
                                                          Get_Character
                    code dec         11           178
                    code dec         15           248     Procedure  &
                                                          specification Advance
                    code dec         15           248
                    code dec         17           272     Procedure Llmain

Character   <<Type>>
     declared as => *** UNKNOWN ***
                    declaration     ** LIBRARY **          Package Standard
                    code ref          6           123     Procedure Llprttoken
                    code dec         11           178     Procedure  &
                                                          Get_Character
                    code dec         18           298     Procedure Readgram

File_Type   <<Type>>
     declared as =>  limited private
                    declaration     ** LIBRARY **          Package Text_Io
                    code dec         18           299     Procedure Readgram

Integer   <<Type>>
     declared as =>  (Implementationdefined)
                    declaration     ** LIBRARY **          Package Standard
                    code dec          2            35     Procedure Ll_Compile
                    code dec          2            36
                    code dec          3            52
                    code dec          3            53
                    code dec          3            54
                    code dec          3            56
                    code dec          3            61
                    code dec          3            62
                    code dec          3            70
                    code dec          3            71
                    code dec          4            83     Function Llfind
                    code dec          4            84
                    code dec         12           195     Function Make_Token
                    code dec         16           266     Procedure Lltakeaction
                    code dec         17           275     Procedure Llmain
                    code dec         17           276
                    code dec         17           280
                                     ...
```

Figure 28-4. ADADL Type Cross Reference Report

PAGE 63

DECLARATION TREE

PAGE NO.	LINE NO.	
1	24	Procedure Ll_Compile
3	79	| Procedure specification Llnexttoken
4	83	| Function Llfind
5	106	| Procedure Llprtstring
6	117	| Procedure Llprttoken
7	128	| Procedure Llskiptoken
8	140	| Procedure Llskipnode
9	153	| Procedure Llskipboth
10	167	| Procedure Llfatal
11	178	| Procedure Get_Character
12	193	| Function Make_Token
13	198	| | Function Cvt_String
15	247	| Package Ll_Tokens
15	248	| | Procedure specification Advance
15	252	| Package body Ll_Tokens
16	255	| Procedure Llnexttoken
16	266	| Procedure Lltakeaction
17	269	| Procedure Llmain
18	297	| | Procedure Readgram
19	302	| | | Procedure Buildright
21	358	| | | Procedure Buildselect
23	407	| | Procedure Parse
24	412	| | | Procedure Erase
24	428	| | | Procedure specification Testsynch
25	431	| | | Procedure Expand
26	436	| | | | Function Match
28	489	| | | Procedure Testsynch
28	490	| | | | Procedure Synchronize

Figure 28-5. ADADL Declaration Tree

```
                                                                    PAGE 64
                                INVOCATION TREE
================================================================================
PAGE NO.    LINE NO.

    1           24      Procedure Ll_Compile
   17          269      | Procedure Llmain
   18          297      | | Procedure Readgram
 ** LIBRARY **          | | | Procedure specification Open
 ** LIBRARY **          | | | Procedure specification Get
 ** LIBRARY **          | | | Procedure specification Skip_Line
   19          302      | | | Procedure Buildright
 ** LIBRARY **          | | | | Procedure specification Get
 ** LIBRARY **          | | | | Procedure specification Put
 ** LIBRARY **          | | | | Function specification End_Of_Line
 ** LIBRARY **          | | | | Procedure specification Skip_Line
 ** LIBRARY **          | | | | Procedure specification Put_Line
   21          358      | | | Procedure Buildselect
 ** LIBRARY **          | | | | Procedure specification Get
 ** LIBRARY **          | | | | Procedure specification Skip_Line
 ** LIBRARY **          | | | Procedure specification Close
   23          407      | | Procedure Parse
    4           83      | | | Function Llfind
   16          255      | | | Procedure Llnexttoken
   16          266      | | | Procedure Lltakeaction
   28          489      | | | Procedure Testsynch
   10          167      | | | | Procedure Llfatal
 ** LIBRARY **          | | | | | Procedure specification Put
    6          117      | | | | | Procedure Llprttoken
    5          106      | | | | | | Procedure Llprtstring
 ** LIBRARY **          | | | | | | | Procedure specification Put
 ** LIBRARY **          | | | | | | Procedure specification Put
 ** LIBRARY **          | | | | | Procedure specification Put_Line
   28          490      | | | | Procedure Synchronize
 ** LIBRARY **          | | | | | Procedure specification Put
    6          117      | | | | | Procedure Llprttoken
    5          106      | | | | | | Procedure Llprtstring
 ** LIBRARY **          | | | | | | | Procedure specification Put
 ** LIBRARY **          | | | | | | Procedure specification Put
    5          106      | | | | | Procedure Llprtstring
 ** LIBRARY **          | | | | | | Procedure specification Put
 ** LIBRARY **          | | | | | Procedure specification Put_Line
   16          266      | | | | | Procedure Lltakeaction
   16          255      | | | | | Procedure Llnexttoken
   25          431      | | | | Procedure Expand
   26          436      | | | | | Function Match
 ** LIBRARY **          | | | | | Procedure specification Put_Line
   10          167      | | | | | Procedure Llfatal
 ** LIBRARY **          | | | | | | Procedure specification Put
    6          117      | | | | | Procedure Llprttoken
    5          106      | | | | | | Procedure Llprtstring
 ** LIBRARY **          | | | | | | | Procedure specification Put
                                ...
```

Figure 28-6. ADADL Invocation Tree

```
                                                                    PAGE 67
                              WITH HIERARCHY
================================================================================
PAGE NO.    LINE NO.

    1           24        Procedure Ll_Compile
  ** LIBRARY **           |  Package Ll_Declarations
  ** LIBRARY **           |  Instantiated Package Integer_Text_Io
  ** LIBRARY **           |  Package Text_Io




                                                                    PAGE 68
                         INTERRUPT CROSS REFERENCE REPORT
================================================================================
                     DEC/REF          LOCATION OF DECLARATION OR REFERENCE
                                   PAGE NO.    LINE NO.    ENCLOSING PROGRAM UNIT

               NO INTERRUPTS TO REFERENCE




                                                                    PAGE 69
                           GENERIC INSTANTIATION REPORT
================================================================================
                                   LOCATION OF DECLARATION OR INSTANTIATION
                                PAGE NO.    LINE NO.    ENCLOSING/INSTANTIATED UNIT

NO GENERICS TO REPORT
                                                                    PAGE 70
```

Figure 28-7. ADADL Additional Cross Reference Reports

```
                        EXCEPTION CROSS REFERENCE REPORT
================================================================================
                    DEC/REF          LOCATION OF DECLARATION OR REFERENCE
                                   PAGE NO.    LINE NO.    ENCLOSING PROGRAM UNIT

Parsing_Error    <<Exception>>
   ** No Dictionary Definition Given **
                    declaration        2          26      Procedure Ll_Compile
             raised code ref          10         174      Procedure Llfatal
             raised code ref          19         334      Procedure Buildright
             raised code ref          20         352
             raised code ref          29         532      Procedure Synchronize
```



```
                                PRAGMA REPORT
================================================================================
                        LOCATION OF REFERENCE
                   PAGE NO.    LINE NO.    ENCLOSING PROGRAM UNIT

                   NO PRAGMAS USED
```



```
                        PROGRAM UNIT RENAMES REPORT
================================================================================
                        LOCATION OF RENAMING DECLARATION
                   PAGE NO.    LINE NO.    ENCLOSING PROGRAM UNIT

NO PROGRAM UNIT RENAMES TO REPORT
```

Figure 28-7 continued: ADADL Additional Cross Reference Reports

```
                                                                    PAGE 73
                          COMPLEXITY SUMMARY REPORT
================================================================================
     Mc Cabe          ADADL
    COMPLEXITY       COMPLEXITY
  code    design     code    design     Line no    Program Unit Name

    10       1         13        1          302    Buildright Procedure
**** WARNING: the code complexity measure for this module is above the  &
   maximum level 10
     2       1          2        1          358    Buildselect Procedure
     3       1          3        1          198    Cvt_String Function
     3       1          3        1          412    Erase Procedure
     7       1         11        1          431    Expand Procedure
**** WARNING: the code complexity measure for this module is above the  &
   maximum level 10
     3       1          3        1          178    Get_Character Procedure
     1       1          1        1           24    Ll_Compile Procedure
     1       1          1        1          252    Ll_Tokens Package body
     1       1          1        1          167    Llfatal Procedure
     5       1          6        1           83    Llfind Function
     1       1          1        1          269    Llmain Procedure
     2       1          2        1          255    Llnexttoken Procedure
     3       1          3        1          106    Llprtstring Procedure
     2       1          2        1          117    Llprttoken Procedure
     1       1          1        1          153    Llskipboth Procedure
     1       1          1        1          140    Llskipnode Procedure
     1       1          1        1          128    Llskiptoken Procedure
     1       1          1        1          266    Lltakeaction Procedure
    11       1         11        1          193    Make_Token Function
**** WARNING: the code complexity measure for this module is above the  &
   maximum level 10
     4       1          5        1          436    Match Function
    11       1         13        1          407    Parse Procedure
**** WARNING: the code complexity measure for this module is above the  &
   maximum level 10
     6       1          6        1          297    Readgram Procedure
    10       1         16        1          490    Synchronize Procedure
**** WARNING: the code complexity measure for this module is above the  &
   maximum level 10
     3       1          3        1          489    Testsynch Procedure
```

Figure 28-8. ADADL Complexity Summary Report

```
                                                                    PAGE 74
                          PROGRAM UNIT ID REPORT
================================================================================
PROGRAM UNIT ID              PROGRAM UNIT OR CONFIGURATION ITEM

  NO PROGRAM UNITS WITH PROGRAM UNIT IDS
```

Figure 28-9. ADADL Program Unit ID Report

```
                                                                    PAGE 75
                  REPORT ON OBJECTS DECLARED BUT NOT USED
================================================================================
                  DEC              LOCATION OF DECLARATION
                                 PAGE NO.   LINE NO.   ENCLOSING PROGRAM UNIT

Caseindex   <<In Parameter>>
     declared as => in Integer
                  declaration      16         266     Procedure Lltakeaction

More   <<In Parameter>>
     declared as => in Boolean := True
                  declaration      11         178     Procedure &
                                                      Get_Character

Tableindex   <<Object>>
     declared as => Integer
                  declaration      19         304     Procedure Buildright
```

Figure 28-10. ADADL Objects Declared but Not Used Report

```
                                                                    PAGE 76
                  REPORT ON TYPES DECLARED BUT NOT USED
================================================================================
                  DEC              LOCATION OF DECLARATION
                                 PAGE NO.   LINE NO.   ENCLOSING PROGRAM UNIT

  NOTHING TO REPORT
```

Figure 28-11. ADADL Types Declared But Not Used Report

PAGE 77

REPORT ON PROGRAM UNITS DECLARED BUT NOT USED

	DEC	LOCATION OF DECLARATION PAGE NO.	LINE NO.	ENCLOSING PROGRAM UNIT
Advance <<Procedure specification>>	declaration	15	248	Package Ll_Tokens
Get_Character <<Procedure>>	declaration	11	178	Procedure Ll_Compile
Ll_Compile <<Procedure>>	declaration	1	24	
Ll_Tokens <<Package body>>	declaration	15	247	Procedure Ll_Compile
	declaration	15	252	Procedure Ll_Compile
Ll_Tokens <<Package>>	declaration	15	247	Procedure Ll_Compile
Llskipboth <<Procedure>>	declaration	9	153	Procedure Ll_Compile
Llskipnode <<Procedure>>	declaration	8	140	Procedure Ll_Compile
Llskiptoken <<Procedure>>	declaration	7	128	Procedure Ll_Compile
Make_Token <<Function>>	declaration	12	193	Procedure Ll_Compile

Figure 28-12. ADADL Program Units Declared But Not Used Report

```
                                                                    PAGE 102
            REPORT ON PROGRAM UNITS WITH HIGH COMPLEXITY METRICS
================================================================================
LINE NO      DESIGN COMPLEXITY      CODE COMPLEXITY      PROGRAM UNIT NAME

302                                      13             Buildright  <<       &
                                                        Procedure >>
431                                      11             Expand  <<           &
                                                        Procedure >>
193                                      11             Make_Token  <<       &
                                                        Function >>
407                                      13             Parse  << Procedure &
                                                         >>
490                                      16             Synchronize  <<      &
                                                        Procedure >>
```

Figure 28-13. ADADL Program Units with High Complexity Metrics Report

```
                                                                    PAGE 103
                        ERROR CROSS REFERENCE REPORT
================================================================================
PAGE NO.    LINE NO.              ERROR MESSAGE

   73                             104 ***** WARNING: Code complexity measure  &
                                   for a module is above the maximum level.

   73                             104 ***** WARNING: Code complexity measure  &
                                   for a module is above the maximum level.

   73                             104 ***** WARNING: Code complexity measure  &
                                   for a module is above the maximum level.

   73                             104 ***** WARNING: Code complexity measure  &
                                   for a module is above the maximum level.

   73                             104 ***** WARNING: Code complexity measure  &
                                   for a module is above the maximum level.
```

Figure 28-14. ADADL Error Cross Reference Report

```
*****************************************************************************
* Testing all paths of  Subprogram: Llfind
*****************************************************************************
-----------------------------------------------------------
 Test conditions case 1 of 4 for subprogram: Llfind

Test conditions required for test case 1 are:

 90: Set (Item < Llsymboltable(Midpoint).Key  ) to False
 92: Set (Item = Llsymboltable(Midpoint).Key  ) to False
-----------------------------------------------------------
Statements to be executed during test case 1 are:

 83:  Procedure Llfind is
 85:  Begin
 86:   Low := 1;
 87:   High := Lltablesize + 1;
 88:  While Low /= High   loop
 89:   Midpoint := (High + Low) / 2;
 90:  If Item < Llsymboltable(Midpoint).Key   then
          *** Condition is False
 92:  Elsif Item = Llsymboltable(Midpoint).Key
          *** Condition is False
 98:  Else
 99:   Low := Midpoint + 1;
 100:  End if    -- for 90
 101:  End Loop
*** Exit loop at 88 when (Low /= High  ) is false.
 102:  Return ( 0 );
 103:  End
-----------------------------------------------------------
 Test conditions case 2 of 4 for subprogram: Llfind

Test conditions required for test case 2 are:

 90: Set (Item < Llsymboltable(Midpoint).Key  ) to False
 92: Set (Item = Llsymboltable(Midpoint).Key  ) to True
 93: Set (Llsymboltable(Midpoint).Kind = Which  ) to False
-----------------------------------------------------------
Statements to be executed during test case 2 are:

 83:  Procedure Llfind is
 85:  Begin
 86:   Low := 1;
 87:   High := Lltablesize + 1;
 88:  While Low /= High   loop
 89:   Midpoint := (High + Low) / 2;
 90:  If Item < Llsymboltable(Midpoint).Key   then
          *** Condition is False
 92:  Elsif Item = Llsymboltable(Midpoint).Key
          *** Condition is True
 93:  If Llsymboltable(Midpoint).Kind = Which   then
          *** Condition is False
 95:  Else
```

Figure 28-15. TestGen Test Conditions for Path Testing of LLFIND

```
 96:  Return ( 0 );
 103:  End
-------------------------------------------------------
 Test conditions case 3 of 4 for subprogram: Llfind

Test conditions required for test case 3 are:

 90: Set (Item < Llsymboltable(Midpoint).Key  ) to False
 92: Set (Item = Llsymboltable(Midpoint).Key  ) to True
 93: Set (Llsymboltable(Midpoint).Kind = Which  ) to True
-------------------------------------------------------
Statements to be executed during test case 3 are:

 83:  Procedure Llfind is
 85:  Begin
 86:   Low := 1;
 87:   High := Lltablesize + 1;
 88:  While Low /= High   loop
 89:   Midpoint := (High + Low) / 2;
 90:  If Item < Llsymboltable(Midpoint).Key   then
          *** Condition is False
 92:  Elsif Item = Llsymboltable(Midpoint).Key
          *** Condition is True
 93:  If Llsymboltable(Midpoint).Kind = Which  then
          *** Condition is True
 94:  Return ( Midpoint );
 103:  End
-------------------------------------------------------
 Test conditions case 4 of 4 for subprogram: Llfind


Test conditions required for test case 4 are:

 90: Set (Item < Llsymboltable(Midpoint).Key  ) to True
-------------------------------------------------------
Statements to be executed during test case 4 are:

 83:  Procedure Llfind is
 85:  Begin
 86:   Low := 1;
 87:   High := Lltablesize + 1;
 88:  While Low /= High   loop
 89:   Midpoint := (High + Low) / 2;
 90:  If Item < Llsymboltable(Midpoint).Key   then
          *** Condition is True
 91:   High := Midpoint;
 100:  End if    -- for 90
 101:  End Loop
*** Exit loop at 88 when (Low /= High  ) is false.
 102:  Return ( 0 );
 103:  End
```

Figure 28-15 continued: TestGen Test Conditions for Path Testing of LLFIND

Test Case Effort Report

page 1 of 2
Number of Test Cases Required for

Module Name	Basis Testing	Branch Testing	Full Path Testing
Ll_Compile	1	1	1
Llfind	4	4	4
Llprtstring	1	2	2
Llprttoken	2	2	2
Llskiptoken	1	1	1
Llskipnode	1	1	1
Llskipboth	1	1	1
Llfatal	1	1	1
Get_Character	3	3	3
Make_Token	11	7	35
Cvt_String	2	2	2
Llnexttoken	2	2	2
Llmain	1	1	1

Test Case Effort Report

page 2 of 2
Number of Test Cases Required for

Module Name	Basis Testing	Branch Testing	Full Path Testing
Readgram	2	2	2
Buildright	9	7	22
Buildselect	1	1	1
Parse	10	8	32
Erase	2	2	2
Expand	6	4	10
Match	3	3	3
Testsynch	2	2	2
Synchronize	6	4	10

Figure 28-16. TestGen Test Case Effort Report

```
Unreachable Statement Report for module: Ll_Compile
 All statements can be reached.

Unreachable Statement Report for module: Llfind
 97:  End if    -- for 93
 There were 1 statements that could not be reached

Unreachable Statement Report for module: Llprtstring
 All statements can be reached.

Unreachable Statement Report for module: Llprttoken
 All statements can be reached.

Unreachable Statement Report for module: Llskiptoken
 All statements can be reached.

Unreachable Statement Report for module: Llskipnode
 All statements can be reached.

Unreachable Statement Report for module: Llskipboth
 All statements can be reached.

Unreachable Statement Report for module: Llfatal
 All statements can be reached.

Unreachable Statement Report for module: Get_Character
 All statements can be reached.

Unreachable Statement Report for module: Make_Token
 All statements can be reached.

Unreachable Statement Report for module: Cvt_String
 All statements can be reached.

Unreachable Statement Report for module: Llnexttoken
 All statements can be reached.

Unreachable Statement Report for module: Llmain
 All statements can be reached.

Unreachable Statement Report for module: Readgram
 All statements can be reached.

Unreachable Statement Report for module: Buildright
 All statements can be reached.

Unreachable Statement Report for module: Buildselect
 All statements can be reached.

Unreachable Statement Report for module: Parse
 All statements can be reached.
                                        ...
```

Figure 28-17. TestGen Unreachable Statement Report for LL_COMPILE

McCabe Cyclomatic Complexity Report

page 1 of 2

Module Name	Design	Code
Ll_Compile	1	1
Llfind	1	5
Llprtstring	1	3
Llprttoken	1	2
Llskiptoken	1	1
Llskipnode	1	1
Llskipboth	1	1
Llfatal	1	1
Get_Character	1	3
Make_Token	1	11
Cvt_String	1	3
Llnexttoken	1	2
Llmain	1	1
Readgram	1	6
Buildright	1	10

McCabe Cyclomatic Complexity Report

page 2 of 2

Module Name	Design	Code
Buildselect	1	2
Parse	1	11
Erase	1	3
Expand	1	7
Match	1	4
Testsynch	1	3
Synchronize	1	9

Figure 28-18. TestGen McCabe Complexity Report for LL_COMPILE

Test Coverage Summary

Page 1 of 3

Module Name	Calls	Stmts Not Done	Complete Path	Branch Path	Basis Path
Ll_Compile	1	0	1/1 (100%)	1/1 (100%)	1/1 (100%)
Llfind	198	4	3/4 (75%)	3/4 (75%)	3/4 (75%)
Llprtstring	0	10	0/2 (0%)	0/2 (0%)	0/1 (0%)
Llprttoken	0	10	0/2 (0%)	0/2 (0%)	0/2 (0%)
Llskiptoken	0	11	0/1 (0%)	0/1 (0%)	0/1 (0%)
Llskipnode	0	12	0/1 (0%)	0/1 (0%)	0/1 (0%)
Llskipboth	0	13	0/1 (0%)	0/1 (0%)	0/1 (0%)
Llfatal	0	10	0/1 (0%)	0/1 (0%)	0/1 (0%)
Get_Character	865	1	3/3 (100%)	3/3 (100%)	3/3 (100%)
Make_Token	133	9	5/35 (14%)	5/7 (71%)	1/11 (9%)

Test Coverage Summary

Page 2 of 3

Module Name	Calls	Stmts Not Done	Complete Path	Branch Path	Basis Path
Cvt_String	133	1	2/2 (100%)	2/2 (100%)	2/2 (100%)
Llnexttoken	134	1	2/2 (100%)	2/2 (100%)	2/2 (100%)
Llmain	1	1	1/1 (100%)	1/1 (100%)	1/1 (100%)
Readgram	1	1	1/2 (50%)	2/2 (100%)	1/2 (50%)
Buildright	64	7	8/22 (36%)	5/7 (71%)	3/9 (33%)
Buildselect	64	1	1/1 (100%)	1/1 (100%)	1/1 (100%)
Parse	1	12	5/32 (16%)	2/8 (25%)	1/10 (10%)
Erase	398	1	2/2 (100%)	2/2 (100%)	2/2 (100%)
Expand	254	5	5/10 (50%)	2/4 (50%)	2/6 (33%)
Match	254	4	1/3 (33%)	1/3 (33%)	1/3 (33%)

Test Coverage Summary

Page 3 of 3

Module Name	Calls	Stmts Not Done	Complete Path	Branch Path	Basis Path
Testsynch	0	12	0/2 (0%)	0/2 (0%)	0/2 (0%)
Synchronize	0	44	0/10 (0%)	0/4 (0%)	0/6 (0%)
Ll_Tokens	0	3	0/1 (0%)	0/1 (0%)	0/1 (0%)
Current_Symbol	133	1	1/1 (100%)	1/1 (100%)	1/1 (100%)
Advance	134	7	5/8 (62%)	6/8 (75%)	5/8 (62%)
Scan_Pattern	257	129	16/585 (3%)	12/40 (30%)	1/50 (2%)
Char_Advance	780	3	1/3 (33%)	1/3 (33%)	1/3 (33%)
Look_Ahead	39	2	1/3 (33%)	1/3 (33%)	1/3 (33%)
Lltakeaction	230	77	32/68 (47%)	32/68 (47%)	32/68 (47%)

Figure 28-19. TestGen Test Coverage Summary using test1.lex

TestGen Sub-Program Invocation Count Report

Module Name	Invocations
Ll_Compile	1
Llfind	198
Llprtstring	0
Llprttoken	0
Llskiptoken	0
Llskipnode	0
Llskipboth	0
Llfatal	0
Get_Character	865
Make_Token	133
Cvt_String	133
Llnexttoken	134
Llmain	1
Readgram	1
Buildright	64
Buildselect	64
Parse	1
Erase	398
Expand	254
Match	254
Testsynch	0
Synchronize	0
Ll_Tokens	0
Current_Symbol	133
Advance	134
Scan_Pattern	257
Char_Advance	780
Look_Ahead	39
Lltakeaction	230

Figure 28-20. TestGen Sub-Program Invocation Count Report using test1.lex

```
************************************************************

Statement Execution Report for Module: Advance


      :
      :   procedure ADVANCE(EOS: out BOOLEAN;
      :     NEXT: out LLTOKEN;
    134:    MORE: in BOOLEAN := TRUE) is
    134:  begin
    134:    EOS := FALSE;
    134:    loop
    257:      SCAN_PATTERN;
    257:      case CUR_PATTERN is
      1:        when END_OF_INPUT =>
      1:          EOS := TRUE;
      1:          return;
     45:        when END_OF_LINE => null;
      9:        when Character_Literal =>
      9:          NEXT := MAKE_TOKEN( CHAR, CURRENT_SYMBOL, CUR_LINE_NUM);
      9:          return;
     78:        when Comment | White_Space => null;
     64:        when Delimiter | Number | Special_Symbol =>
     64:          NEXT := MAKE_TOKEN( LIT, CURRENT_SYMBOL, CUR_LINE_NUM);
     64:          return;
     60:        when Identifier =>
     60:          NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM);
     60:          return;
      0:        when String_Literal =>
      0:          NEXT := MAKE_TOKEN( STR, CURRENT_SYMBOL, CUR_LINE_NUM);
      0:          return;
      0:        when others =>
      0:          NEXT := MAKE_TOKEN( LIT, CURRENT_SYMBOL, CUR_LINE_NUM);
      0:          return;
    123:      end case;
    123:    end loop;
    257:  end ADVANCE;

************************************************************
```

Figure 28-21. TestGen Statement Execution Report using test1.lex for ADVANCE

```
************************************************************
Branch Path Coverage Analysis for Module: Advance


There were 2 paths not tested:

*****************************************************************************
* Testing all statements of  Subprogram: Advance
*****************************************************************************
-------------------------------------------------------
 Test conditions case 1 of 2 for subprogram: Advance

Test conditions required for test case 1 are:
 51: Set (Cur_Pattern ) to String_Literal
-------------------------------------------------------
Statements to be executed during test case 1 are:

 46:  Procedure Advance is
 47:  Begin
 48:   Eos := False;
 49:  Loop
 50:   Scan_Pattern;
 51:  Case Cur_Pattern  is
          *** Case variable is String_Literal
 66:  When String_Literal  =>
 67:   Next := Make_Token( Str, Current_Symbol, Cur_Line_Num);
 68:  Return ;
 74:  End
-------------------------------------------------------
 Test conditions case 2 of 2 for subprogram: Advance

Test conditions required for test case 2 are:
 51: Set (Cur_Pattern ) to others
-------------------------------------------------------
Statements to be executed during test case 2 are:

 46:  Procedure Advance is
 47:  Begin
 48:   Eos := False;
 49:  Loop
 50:   Scan_Pattern;
 51:  Case Cur_Pattern  is
          *** Case variable is others
 69:  When others  =>
 70:   Next := Make_Token( Lit, Current_Symbol, Cur_Line_Num);
 71:  Return ;
 74:  End

2 of 8 paths were not tested.
This module is 75 percent tested.

************************************************************
```

Figure 28-22. TestGen Branch Path Coverage Analysis using test1.lex for ADVANCE

```
**************************************************:*********
Structured Testing Path Coverage Analysis for Module: Advance


There were 3 paths not tested :

*************************************************************************
* Testing a Structured path selection of  Subprogram: Advance
*************************************************************************
-------------------------------------------------------
 Test conditions case 1 of 3 for subprogram: Advance


Test conditions required for test case 1 are:

 51: Set (Cur_Pattern ) to End_Of_Input
-------------------------------------------------------
Statements to be executed during test case 1 are:

 46:  Procedure Advance is
 47:  Begin
 48:   Eos := False;
 49:  Loop
 50:   Scan_Pattern;
 51:  Case Cur_Pattern  is
         *** Case variable is End_Of_Input
 52:  When End_Of_Input  =>
 53:   Eos := True;
 54:  Return ;
 74:  End
-------------------------------------------------------
 Test conditions case 2 of 3 for subprogram: Advance


Test conditions required for test case 2 are:

 51: Set (Cur_Pattern ) to String_Literal
-------------------------------------------------------
Statements to be executed during test case 2 are:

 46:  Procedure Advance is
 47:  Begin
 48:   Eos := False;
 49:  Loop
 50:   Scan_Pattern;
 51:  Case Cur_Pattern  is
         *** Case variable is String_Literal
 66:  When String_Literal  =>
 67:   Next := Make_Token( Str, Current_Symbol, Cur_Line_Num);
 68:  Return ;
 74:  End
```

Figure 28-23. TestGen Structured Testing Path Coverage Analysis using test1.lex for ADVANCE

```
Test conditions required for test case 3 are:

 51: Set (Cur_Pattern ) to others
-------------------------------------------------------
Statements to be executed during test case 3 are:

 46:  Procedure Advance is
 47:  Begin
 48:   Eos := False;
 49:  Loop
 50:   Scan_Pattern;
 51:  Case Cur_Pattern  is
         *** Case variable is others
 69:  When others  =>
 70:   Next := Make_Token( Lit, Current_Symbol, Cur_Line_Num);
 71:  Return ;
 74:  End

3 of 8 paths were not tested.
This module is 62 percent tested.

************************************************************
```

Figure 28-23 continued: TestGen Structured Testing Path Coverage Analysis using test1.lex for ADVANCE

Test Coverage Summary

Page 1 of 3

Module Name	Calls	Stmts Not Done	Complete Path	Branch Path	Basis Path
Ll_Compile	2	0	1/1 (100%)	1/1 (100%)	1/1 (100%)
Llfind	510	2	4/4 (100%)	4/4 (100%)	4/4 (100%)
Llprtstring	0	10	0/2 (0%)	0/2 (0%)	0/1 (0%)
Llprttoken	0	10	0/2 (0%)	0/2 (0%)	0/2 (0%)
Llskiptoken	0	11	0/1 (0%)	0/1 (0%)	0/1 (0%)
Llskipnode	0	12	0/1 (0%)	0/1 (0%)	0/1 (0%)
Llskipboth	0	13	0/1 (0%)	0/1 (0%)	0/1 (0%)
Llfatal	0	10	0/1 (0%)	0/1 (0%)	0/1 (0%)
Get_Character	2250	1	3/3 (100%)	3/3 (100%)	3/3 (100%)
Make_Token	353	5	6/35 (17%)	6/7 (86%)	1/11 (9%)

Test Coverage Summary

Page 2 of 3

Module Name	Calls	Stmts Not Done	Complete Path	Branch Path	Basis Path
Cvt_String	353	1	2/2 (100%)	2/2 (100%)	2/2 (100%)
Llnexttoken	355	1	2/2 (100%)	2/2 (100%)	2/2 (100%)
Llmain	2	1	1/1 (100%)	1/1 (100%)	1/1 (100%)
Readgram	2	1	1/2 (50%)	2/2 (100%)	1/2 (50%)
Buildright	128	7	8/22 (36%)	5/7 (71%)	3/9 (33%)
Buildselect	128	1	1/1 (100%)	1/1 (100%)	1/1 (100%)
Parse	2	12	5/32 (16%)	2/8 (25%)	1/10 (10%)
Erase	1105	1	2/2 (100%)	2/2 (100%)	2/2 (100%)
Expand	715	5	5/10 (50%)	2/4 (50%)	2/6 (33%)
Match	715	4	1/3 (33%)	1/3 (33%)	1/3 (33%)

Test Coverage Summary

Page 3 of 3

Module Name	Calls	Stmts Not Done	Complete Path	Branch Path	Basis Path
Testsynch	0	12	0/2 (0%)	0/2 (0%)	0/2 (0%)
Synchronize	0	44	0/10 (0%)	0/4 (0%)	0/6 (0%)
Ll_Tokens	0	3	0/1 (0%)	0/1 (0%)	0/1 (0%)
Current_Symbol	353	1	1/1 (100%)	1/1 (100%)	1/1 (100%)
Advance	355	4	5/8 (62%)	7/8 (88%)	5/8 (62%)
Scan_Pattern	712	106	17/585 (3%)	13/40 (32%)	1/50 (2%)
Char_Advance	2018	3	1/3 (33%)	1/3 (33%)	1/3 (33%)
Look_Ahead	123	2	1/3 (33%)	1/3 (33%)	1/3 (33%)
Lltakeaction	659	69	35/68 (51%)	35/68 (51%)	35/68 (51%)

Figure 28-24. TestGen Test Coverage Summary using test1.lex & sample.lex

	C_NEST_ ENTITY	C_UNIQ_ OPERANDS	C_UNIQ_ OPERATORS	C_TOT_ OPERANDS	C_TOT_ OPERATORS	X_HAL_ DIFFICULTY	X_HAL_EST_ EFFORT	X_HAL_EST_ LENGTH
ifind	1	16	21	41	67	26.90625	15138.0205078	156.23866272
lprtstring	1	9	15	20	29	16.6666679382	3744.38623047	87.1326828003
lprttoken	1	13	13	22	28	11	2585.24169922	96.2114334106
lskiptoken	1	14	9	20	23	6.42857122421	1250.44177246	81.8322906494
lskipnode	1	19	10	30	33	7.89473676682	2416.20629883	113.929901123
lskipboth	1	20	10	31	34	7.75	2471.84619141	119.657844543
lfatal	1	12	9	18	22	6.75	1185.92578125	71.5488739014
et_Character	1	17	14	29	38	11.9411764145	3963.6484375	122.789840698
vt_String	2	8	16	20	33	20	4860.06005859	88
fake_Token	1	38	22	133	186	38.5	72545.4765625	297.528747559
l_Tokens (Spec	1	10	8	13	15	5.19999980927	607.141052246	57.2192802429
lnextroken	1	13	15	19	29	10.9615383148	2529.40844727	106.709075928
luildright	3	40	22	125	158	34.375	57923.1679688	310.984619141
luildselect	3	16	14	27	33	11.8125	3477.75854492	117.302970886
leadgrm	2	65	23	241	296	42.6384582S2	147900.65625	495.495819092
rase	3	10	14	19	32	13.3000001907	3109.98022461	86.5222473145
fatch	4	14	15	25	43	13.3928575516	4424.23291016	111.906326294
xpand	3	40	24	124	169	37.2000007629	65397.5976563	322.916229248
ynchronize	4	39	24	115	164	35.3846168518	59009.546875	316.169799805
easynch	3	42	24	132	193	37.7142829895	74087	336.516418457
me	2	81	34	390	552	81.851852417	527817.6875	686.501586914

Figure 28-25. QualGen Report Excerpt

C_HAL_LENGTH	X_HAL_EST_LEVEL	X_HAL_JC	X_HAL_VOCAB	X_HAL_EST_LANG	X_HAL_VOLUME	C_DES_UNIQ_OPERANDS	C_DES_UNIQ_OPERATORS	C_DES_TOT_OPERANDS
108	0.03716608509	20.9104194641	37	0.77715837955	562.62097168	9	6	10
49	0.05999999866	13.4797897339	24	0.80878734589	224.663162231	3	4	4
50	0.09090909362	21.3656368256	26	1.94233059883	235.021987915	1	3	2
43	0.15555556118	30.257604599	23	4.70673894882	194.513168335	1	3	2
63	0.12666666508	38.7666854858	29	4.91044712067	306.05279541	1	3	2
65	0.12903225422	41.1543639038	30	5.31026601791	318.947875977	1	3	2
40	0.14814814925	26.0285491943	21	3.85608124733	175.692703247	1	3	2
67	0.08374384046	27.7971897125	31	2.32784342766	331.931152344	9	6	11
53	0.05000000075	12.1501502991	24	0.60730752687	243.003005981	5	5	6
319	0.02597402595	48.9428062439	60	1.2712417841	1884.2980957	18	6	21
28	0.19230769575	22.4534416199	18	4.31796979904	116.757896423	10	7	13
48	0.09122806787	21.051153183	28	1.92045605183	230.753036499	1	3	2
283	0.02909090929	49.0192756653	62	1.42601525784	1685.03759766	5	5	8
60	0.08465608209	24.9238872528	30	2.10995864868	294.413421631	4	5	6
537	0.02345300466	81.3517837524	88	1.9079438448	3468.71484375	11	5	20
51	0.07518796623	17.5814342499	24	1.32191228867	233.833084106	1	3	2
68	0.074666664	24.6655883789	29	1.84169721603	330.342712402	3	4	4
293	0.02688172087	47.258064427	64	1.27037811279	1758	6	7	11
279	0.02826086991	47.1295547485	63	1.3319221735	1667.66113281	4	5	6
325	0.02651515231	52.0871086121	66	1.38109767437	1964.42810059	5	5	8
942	0.01221719477	78.7819900513	115	0.96249490976	6448.45166016	15	7	28

Figure 28-25 continued: QualGen Report Excerpt

_DES_TOT_ OPERATORS	X_DES_HAL_ DIFFICULTY	X_DES_HAL_ EST_EFFORT	X_DES_HAL_ EST_LENGTH	X_DES_HAL_ LENGTH	X_DES_HAL_ EST_LEVEL	X_DES_HAL_ IC	X_DES_HAL_ VOCAB
11	3.333333325386	273.482330322	44.0391006471	21	0.300000011921	24.6134109497	15
4	2.666666746140	59.8902397156	12.7548875809	8	0.375	8.422064781190	7
3		3	30	4.754887580870	5	0.333333343270	3.333333492280	4
3		3	30	4.754887580870	5	0.333333343270	3.333333492280	4
3		3	30	4.754887580870	5	0.333333343270	3.333333492280	4
3		3	30	4.754887580870	5	0.333333343270	3.333333492280	4
11	3.666666507720	315.155822754	44.0391006471	22	0.272727280860	23.4413433075	15
6		3	119.589408875	23.2192802429	12	0.333333343270	13.2877120972	10
25	3.499999761580	738.178955078	90.5684280396	46	0.285714294890	60.2595100403	24
14	4.550000190730	502.144805908	52.8707656860	27	0.219780221580	24.2552738190	17
3		3	30	4.754887580870	5	0.333333343270	3.333333492280	4
9		4	225.891113281	23.2192802429	17	0.25	14.1181945801	10
6	3.749999761580	142.646621704	19.6096401215	12	0.266666680570	10.1437606812	9
22	4.545454502110	763.636352539	49.6633872986	42	0.219999998810	36.9599990845	16
3		3	30	4.754887580870	5	0.333333343270	3.333333492280	4
4	2.666666746140	59.8902397156	12.7548875809	8	0.375	8.422064781190	7
12	6.416666984560	546.12322998	35.1612586975	23	0.155844151970	13.2639141083	13
7	3.749999761580	154.533843994	19.6096401215	13	0.266666680570	10.989074707	9
10		4	239.178817749	23.2192802429	18	0.25	14.9486761093	10
36	6.533333301540	1864.63696289	78.2548446655	64	0.153061226010	43.6842269897	22

Figure 28-25 continued: QualGen Report Excerpt

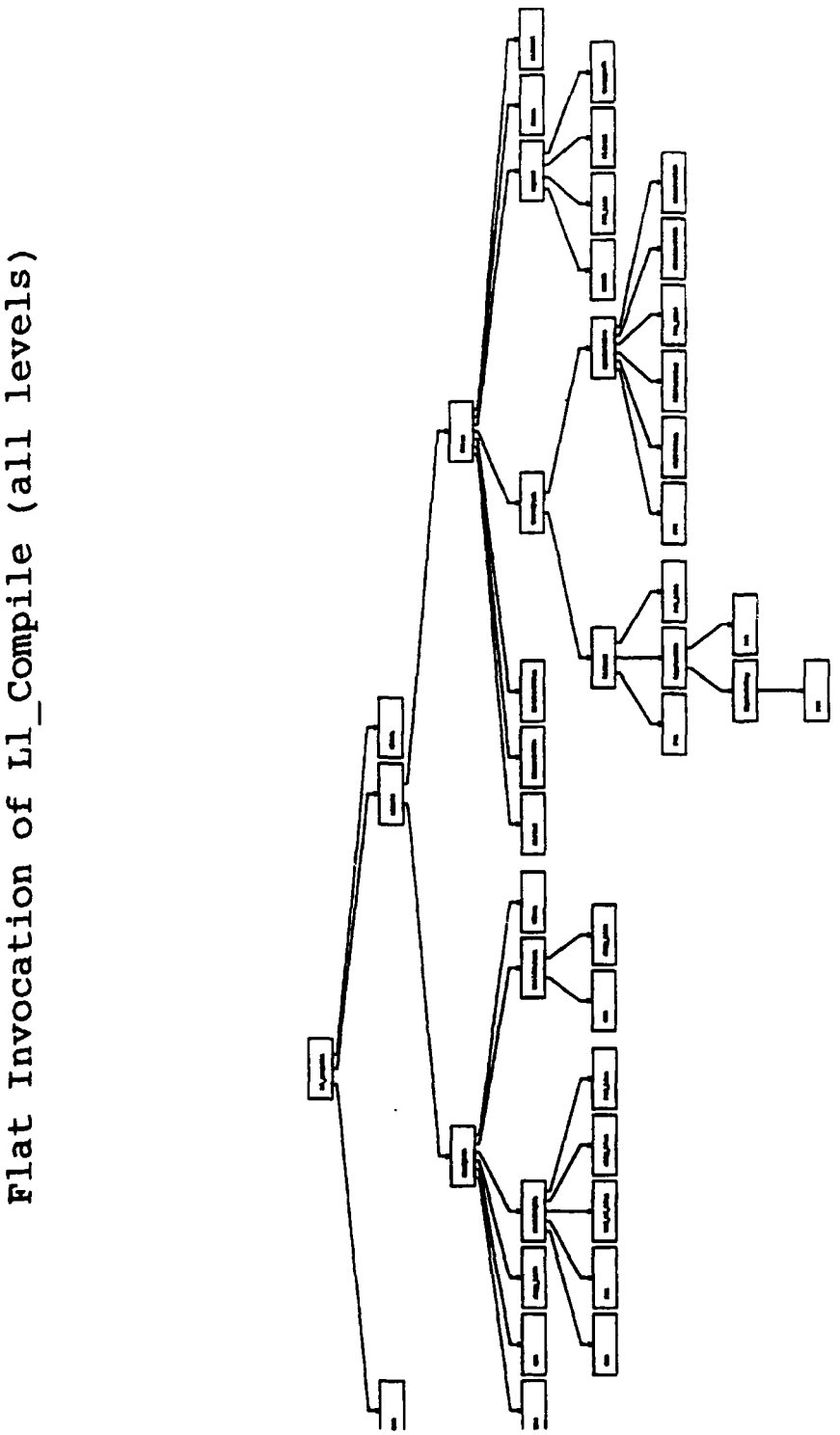


Figure 28-26. GrafBrowse Flat Invocation Graph of LL_COMPILE

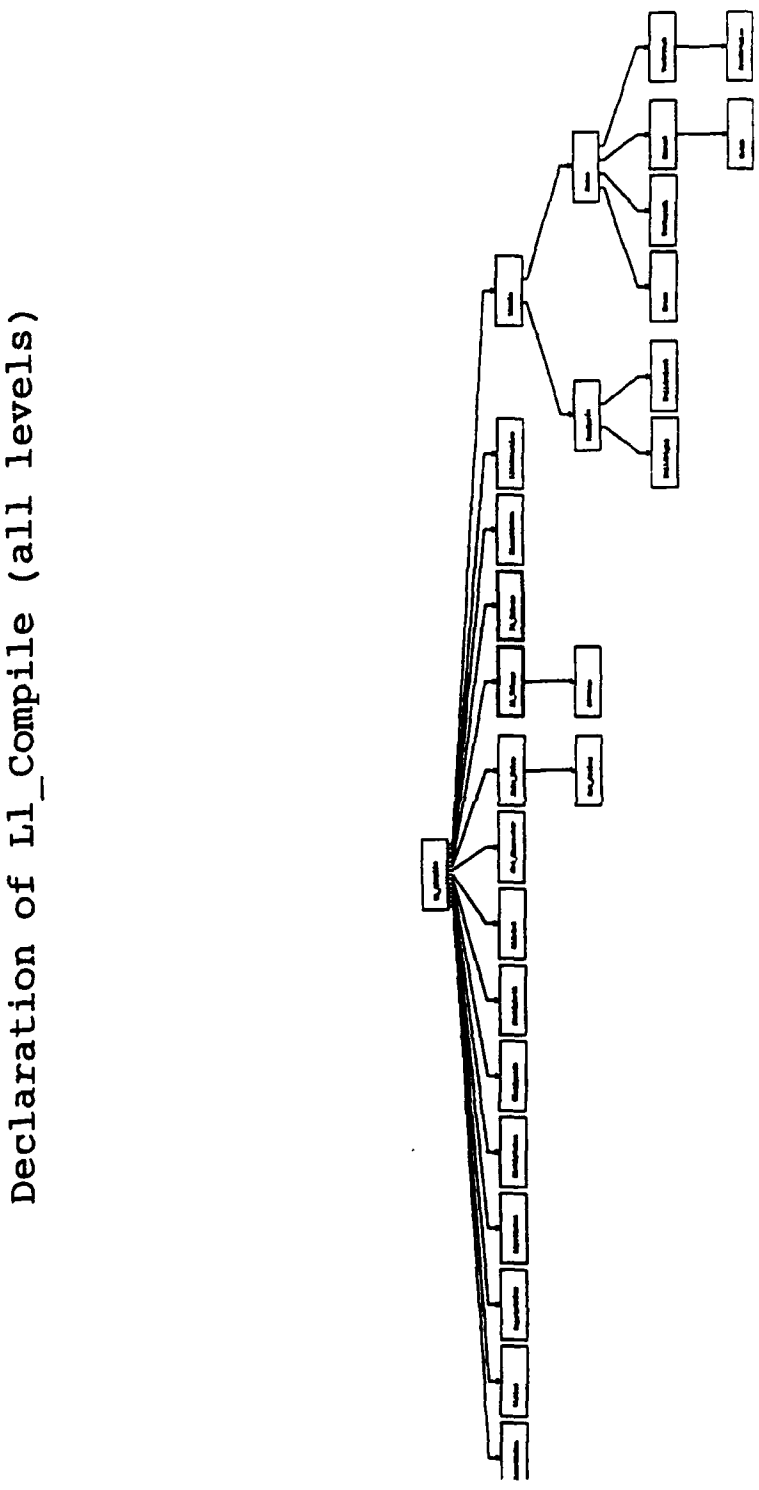


Figure 28-27. GrafBrowse Declaration Tree of LL_COMPILE

Flat Callby of Llfind (all levels)

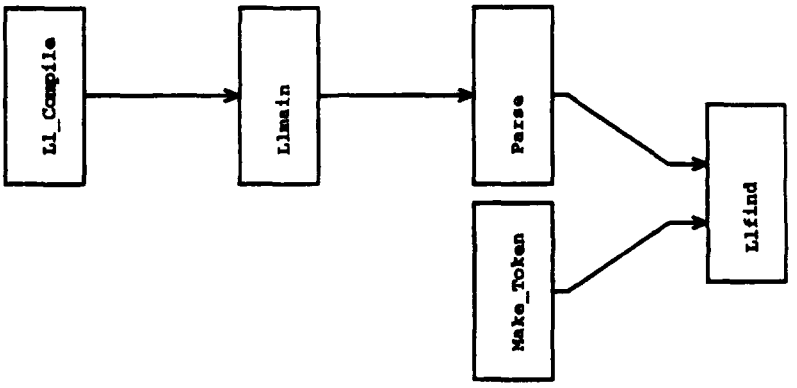


Figure 28-28. GrafBrowse Flat Callby Tree of LLFIND

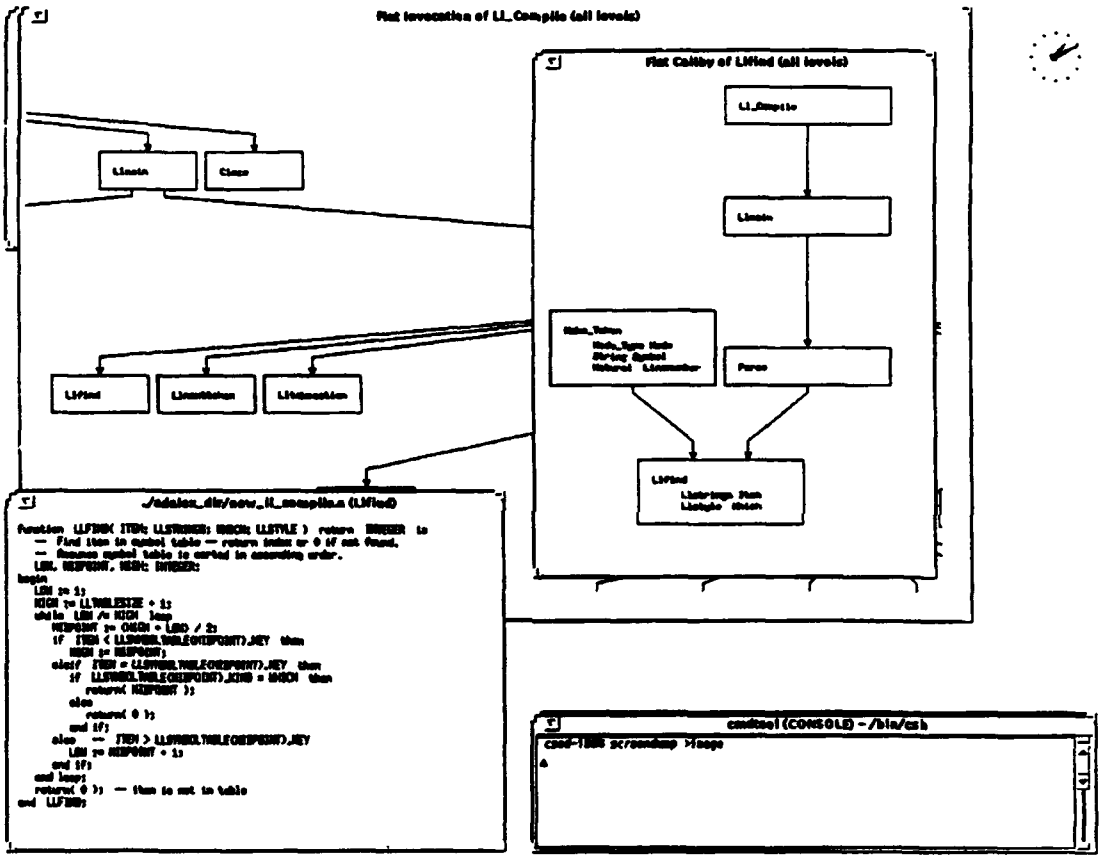


Figure 28-29. Grafbrowse Browsing LLFIND

REFERENCES

[ANSI/IEEE 1983] ANSI/IEEE Standard 829-1983. February 1983. *Standard for Software Test Documentation.* Institute of Electrical and Electronics Engineers, Inc.

[ANSI/MIL 1983] ANSI/Military Standard 1815A. January 1983. *Ada Programming Language.*

[AFSCP 1986] Air Force Systems Command Pamphlet 800-43. January 1986. *Software Management Indicators.* Air Force Systems Command.

[AFSCP 1987] Air Force Systems Command Pamphlet 800-14. January 1987. *Software Quality Indicators.* Air Force Systems Command.

[Boehm 1980] Boehm, B.W. 1980. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall.

[Boehm 1988] Boehm, B.W. and P.N. Papaccio. 1988. "Understanding and Controlling Software Cost." *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, (Oct), pp. 1462-1477.

[DoD-STD-2167A] DoD Standard 2167A. 29 February 1988. *Defense System Software Development.*

[DoD-STD-2168] DoD Standard 2168. 1 August 1986. *Defense System Software Quality Program.*

[DoDI 1991] DoD Instruction 5000.2. 1 January 1991. *Defense Acquisition Management Policies and Procedures.*

[Dunn 1984] Dunn R.H. 1984. *Software Defect Removal.* NY: McGraw-Hill.

[GPALS 1992a] Strategic Defense Initiative Organization. 20 February 1992. *GPALS Software Quality Program Plan (SQPP)*, Annex D to the GPALS CRLCMP. SDI-S-SD-92-000005.

[GPALS 1992b] Strategic Defense Initiative Organization. 30 February 1992. *GPALS Contract Requirements Packages (CRPs) Guidelines for Computer Resource Issues.* SDI-S-SD-92-000005.

[GPALS 1992c] Strategic Defense Initiative Organization. 15 July 1992. *GPALS Software Standards.* SDI-S-SD-91-000003-01.

References

[Graham 1991] Graham, D.R. 1991. "The MD Wants 100% Automated Testing: A Case History." In *Proceedings 8th International Conference on Software Testing*, 17-20 June, Washington, DC.

[Halstead 1977] Halstead, M.H. 1977. *Elements of Software Science*. NY: Elsevier North-Holland Publishing.

[Hennell 1976] Hennell, M.A., M.R. Woodward, and D. Hedley. 1976. "On Program Analysis." *Information Processing Letters*, Vol. 5, No. 5, (Nov):136-140.

[Hook 1991] Hook, A.A. et al. June 1991. *Availability of Ada and C++ Compilers, Tools, Education, and Training*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2601.

[Humphrey 1987] Humphrey, W.S. and W.L. Sweet. September 1987. *A Method for Assessing the Software Engineering Capability of Contractors*. Pittsburgh, PA: Software Engineering Institute. CMU/SEI-87-TR-23.

[IEEE 1987] IEEE Standard P1044. December 1987. *Draft Standard of: A Standard Classification for Software Errors, Faults, and Failures*. Institute of Electrical and Electronics Engineers, Inc., The Standard Classification for Software Errors, Faults, and Failures Working Group of the Software Engineering Standards Subcommittee.

[IEEE 1990] IEEE Standard 610. 10 December 1990. *IEEE Standard Glossary of Software Engineering Terminology*. Institute of Electrical and Electronics Engineers, Inc.

[IEEE 1992] IEEE Standard 1175. 20 August 1992. *A Trial-Use Standard Reference Model for Computing System Tool Interconnections*. Institute of Electrical and Electronics Engineers, Inc.

[KPMG 1992] KPMG Peat Marwick. January 1992. *Software Quality Assurance Survey*. MA: Massachusetts Computer Software Council, Inc.

[Korel 1991] Korel, B. and B. Sherlund. 1991. "Modification Oriented Software Testing." In *Proceedings 8th International Conference on Testing Computer Software*, June 17-20, Washington, DC, pp.143-152.

[Martin Marietta 1991] Martin Marietta. January 1991. *Pro-90 Engineering Handbook, Software Metrics*.

[Martin Marietta 1992] Martin Marietta IS. 11 February 1992. *Technical Report for the Software Metrics Tutorial.* NTB-137-25-02-01.

[McCabe 1976] McCabe, T.J. 1976. "A Complexity Measure." *IEEE: Transactions on Software Engineering*, Vol. 2, No. 4 (Dec), pp. 308-320.

[McCabe 1982] McCabe, T.J. December 1982. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric.* NBS Special Publication 500-99. Gaithersburg, MD: National Institute of Standards and Technology.

[Meeson 1989] Meeson, R.N. 1989. *Ada Lexical Analyzer Generator User's Guide.* Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2109.

[Mohanty 1976] Mohanty, S.N. June 1976. *Automatic Program Testing.* Ph.D. Diss., Polytechnic Institute of New York.

[Mosemann 1992] Mosemann, L.K., II. 1992. "Improving Software Quality Through Measurement." *CrossTalk: The Journal of Defense Software Engineering*, No. 36, (Sept), pp. 2-5.

[Musa 1987] Musa, J.D., A. Iannino, and K. Okumoto. 1987. *Software Reliability Measurement, Prediction, and Application.* NY: McGraw-Hill.

[Paulk 1991] Paulk, M.C., B. Curtis, and M.B. Chrissis. August 1991. *Capability Maturity Model for Software.* Pittsburgh, PA: Software Engineering Institute. CMU/SEI-91-TR-24, ESD-TR-91-24.

[Price 1992a] Price, G., G.T. Daitch, D. Murdok, and E. Hidden. April 1992. *Test Preparation, Execution, and Analysis Tool Report.* Hill Air Force base, UT: U.S. Air Force Software Technology Support Center.

[Price 1992b] Price, G., B. Ragland, D. Murdok, and E. Hidden. April 1992. *Software Test Tool Report - Source Code Static Analysis.* Hill Air Force base, UT: U.S. Air Force Software Technology Support Center.

[RADC 1983] Rome Air Development Center. July 1983. *Software Quality Measurement for Distributed Systems.* RADC-TR-83-175.

[SDIO 1992a] GPALS Computer Resources Working Group, Software Quality Improvement and Standards Committee. January 1992. *Software Metrics Evaluation Plan for the Level 2 System Simulator.*

[SDIO 1992b] Strategic Defense Initiative Organization. March 1992. SDIO Directive No. 3405 (Revision 1). *Strategic Defense Initiative Organization (SDIO) Software Policy.*

[SDIO 1992c] Strategic Defense Initiative Organization. 30 March 1992. *Contract Requirements Packages (CRPs) Guidelines for Computer Resource Issues.* SDI-S-SD-92-000005.

[SPC 1991] Software Productivity Consortium. 1991. *Ada Quality and Style: Guidelines for Professional Programmers.* NY: Van Nostrand Reinhold.

[SQE 1990] Software Quality Engineering. December 1990. *Software Measures and Practices Benchmark Study.* Jacksonville, FL: Software Quality Engineering. TR-900.

[SQE 1991] Software Quality Engineering. 1991. *1990 Testing Practices Survey.* Jacksonville, FL: Software Quality Engineering.

[Sittenauer 1991] Sittenauer, C., G.T. Daitch, D. Samson, D. Dyer, G. Price, and J. Hugie. May 1991. *Software Test Tool Report.* Hill Air Force base, UT: U.S. Air Force Software Technology Support Center.

[U.S. Army 1992] U.S. Army Armament Research, Development and Engineering Center. June 1992. *Software Metrics in Test & Evaluation: AMC Guidance for Implementation of STEP Metrics.* Draft.

[Youngblut 1991] Youngblut, C., B.R. Brykczynski, and R.N. Meeson. October 1991. *An Examination of Selected Commercial Software Testing Tools.* Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2628.

[Yourdon 1990] Yourdon, E. 1990. "Object-Oriented Analysis." In *Proceedings CASE World,* 20-22 March. Los Angeles, CA.

ACRONYMS AND ABBREVIATIONS

ADADL	Ada-based Design and Documentation Language
ADAMAT	Ada Measurement and Analysis Tool
ADW	Application Development Workbench
AFSCP	Air Force Systems Command Pamphlet
AISLE	Ada Integrated Software Lifecycle Environment
ANSI	American National Standards Institute
APSE	Ada Programming Support Environment
ASA	Advanced System Analyzer
ASCII	American Standard Code for Information Interchange
AT&T	Atlantic Telephone and Telegraph
ATEST	Ada Test and Analysis Tools
ATF	Advanced Tactical Fighter
BAT	Battlemap Analysis Tool
BOD	Basic Operating Database
CASE	Computer-aided Software Engineering
CCC	Computer Command & Control Company
CCITT	Consulting Committee on International Telegraphy and Telephon
CMM	Capability Maturity Model
CMS	Critical Metrics Set
COCOMO	Constructive Cost Model
CRP	Contract Requirements Package
CRWG	Computer Resources Working Group

CSCI	Computer Software Component Item
CSU	Computer Software Unit
CUA	Common User Access
CUI	Common User Interface
DoD	Department of Defense
DoDI	Department of Defense Instruction
DDTs	Distributed Defect Tracking System
DEC	Digital Equipment Corporation
GKS	Graphical Kernel System
GPALS	Global Protection Against Limited Strikes
HPGL	Hewlett-Packard Graphics Language
IBM	International Business Machines
ICC	Irvine Compiler Corporation
IDA	Institute for Defense Analyses
IDE	Interactive Development Environments
IEEE	Institute for Electrical and Electronics Engineers, Inc.
IEW	Information Engineering Workbench
IL	Intermediate Language
ISO	International Organization for Standardization
JIAWG	Joint Integrated Avionics Working Group
L2SS	Level 2 System Simulator
LAN	Local Area Network
LCSAJ	Linear Code Sequence and Jump
LDRA	Liverpool Data Research Associates
LH	Light Helicopter
LRM	Language Reference Manual (Ada)

MALPAS	Malvern Program Analysis Suite
MCCR	Mission Critical Computer Resource
MIL	Military
MIS	Management Information System
MOD	Ministry of Defense
NASA	National Aeronautics and Space Administration
NATO	North Atlantic Treaty Organization
NCR	National Cash Register
NTDS	Naval Tactical Data System
NTB	National Test Bed
PC	Personal Computer
PCTE	Portable Common Tool Environment
PDL	Program Design Language
PMM	Process Maturity Model
QA	Quality Assurance
QES	Quality Engineering Software
QUES	Quality Evaluation System
RADC	Rome Air Development Center
RDDTs	Remote Distributed Defect Tracking System
RTP	Rex, Thompson & Partners
StP	Software through Pictures
SAGE	Semiautomated Ground Environment
SASET	Software Architecture, Sizing Tool
SDI	Strategic Defense Initiative
SDIO	Strategic Defense Initiative Organization
SDL	System Development Language

SEI	Software Engineering Institute
SES	Scientific and Engineering Software
SGML	Standard Generalized Markup Language
SMEP	Software Metrics Evaluation Plan
SPC	Software Productivity Consortium
SPCR	Software Problem Change Report
SPS	Software Productivity Solutions
SQA	Software Quality Assurance
SQE	Software Quality Engineering
SQI&S	Software Quality Improvement and Standards
SQMS	Software Quality Management System
SRE	Software Reliability Engineering
SSD	Space Systems Division
STARS	Software Technology for Adaptable, Reliable Systems
START	Structured Testing and Requirements Tool
STD	Standard
STEP	Software Test and Evaluation Panel
STSC	Software Technology Support Center
TBGEN	Test Bed Generator
TC	Testing Comprehensiveness
TCMON	Test Coverage Monitor
TCP	Transmission Control Protocol
TCPOST	Test Coverage Monitor Postprocessor
TDF	Test Data File
TER	Test Effectiveness Ratio
TST	Test Support Tool

UDP	User Datagram Protocol
US	United States
VDM	Vienna Development Method
WIS	Worldwide Military Command and Control System Information System
WWMCCS	Worldwide Military Command and Control System
WR	Work Request
XG&M	Xinotech Guidelines, Standards, and Metrics Analyzer

GLOSSARY

The reader is assumed to be familiar with general software-related terms and, therefore, this glossary focuses on testing and evaluation terms. The reader is referred to the *IEEE Standard Glossary of Software Engineering Terminology* [IEEE 1990] for definitions of additional terms.

Acceptance Testing. Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable a customer to determine whether or not to accept the system.

Assertion. A logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution; for example, "*A* is positive and greater than *B*."

Assertion Testing. A technique which inserts assertions about a program state or the relationship between program variables into the program code. The truth of the assertions is determined as the program executes.

Audit. (1) An independent review for the purpose of assessing compliance with software requirements, specifications, baselines, standards, procedures, instructions, codes, and contractual and licensing agreements. (2) An activity to determine through investigation the adequacy of, and adherence to, established procedures, instructions, specifications, codes, and standards or other applicable contractual and licensing requirements, and the effectiveness of the implementation.

Auditing. Checking for conformance of code to prescribed programing standards and practices.

Basic Execution Time Model. A software reliability model in which the failure process is assumed to be a nonhomogeneous Poisson process with linearly decreasing failure intensity.

Basis Paths. Program paths that have no iteration.

Block. (1) In problem-oriented languages, a computer program subdivision that serves to group related statements, delimit routines, specify storage allocation, delineate the applicability of labels, or segment paths of the computer program for other purposes. (2) A group of contiguous storage locations, computer program statements, records, words, characteristics, or bits that are treated as a unit.

Bottom-up Testing. A systematic testing strategy that seeks to test those modules at the bottom of the invocation structure first. These modules are tested independently using test drivers to invoke them, then modules at the next higher level that call these modules are tested, and so on.

Boundary Value Analysis. A test data selection technique in which test data are chosen to lie along boundaries of the input domain (or output range) classes, data structures, procedure parameters, etc. Choices often include maximum, minimum, and trivial values or parameters.

Branch Testing. Testing designed to execute each outcome of each decision point in a computer program.

Calendar Time. Chronological time, including time during which a computer may not be running.

Call Graph. A diagram that identifies the modules in a system or computer program and shows which modules call one another.

Cause-Effect Graphing. A test data selection technique where the input and output domains are partitioned into classes and analysis is performed to determine what effects are caused by what inputs. A minimum set of inputs is chosen that will cover the entire effect set.

Change Request. A document used to propose, transmit, and record changes to a specification.

Clock Time. Elapsed time from the start to the end of program execution, including wait time, on a running computer.

Code Auditor. An automated tool which checks for conformance to prescribed programming standards and practices.

Code Inspection. See **Inspection**.

Code Review. A meeting at which software code is presented to project personnel, managers, users, customers, or other interested parties for comment or approval.

Comparator. A software tool which compares two computer programs, files, or sets of data to identify commonalities or differences. Typical objects of comparison are similar versions of source code, object code, data base files, or test results.

Complexity. The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces and conditional branches, the degree of nesting, the types of data structures, and other system characteristics.

Component. One of the parts of a system. A component may be hardware or software and may be subdivided into other components.

Concurrent Processes. Processes that may execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process or the occurrence of an external event.

Control Flow. The sequence in which operations are performed during the execution of a computer program.

Control Flow Knot. A control flow knot occurs when two control flow jumps cross. It is defined as: A jump (P, Q) and another jump (A, B) give rise to a knot if (1) P lies within (A, B) and Q lies outside, or (2) Q lies within (A, B) and P lies outside. Variations on basic knots include down-down knots, up-down knots, and up-up knots.

Correctness. See **Program Correctness**.

Coverage Analyzer. A software tool which determines and assesses measures associated with the invocation of program structural elements to determine the adequacy of a test run.

Coverage Measure. In general, a measure of the testing coverage achieved as a result of a test, often expressed as a percentage of the number of statements, branches, or paths that were traversed.

Cross-Referencer. (1) A computer program that provides cross-reference information on system components. For example, programs can be cross-referenced with other programs,

macros, and parameter names. This capability is useful in assessing the impact of changes to one area or another. (2) A utility program which provides cross-reference data concerning a program written in a higher level language. These utility programs analyze a source program and provide as output such data as follows: 1. Statement label cross-index, 2. Data name cross-index, 3. Literal usage cross-index, 4. Inter-subroutine call cross-index, 5. Statistical counts of statement types.

Cyclomatic Complexity. A measure of program complexity derived from the control graph of a program. The cyclomatic complexity of a program is equivalent to the number of decision statements plus 1.

Data Flow. The sequence in which data transfer, use, and transformation are performed during the execution of a computer program.

Data Flow Analysis. Consists of the graphical analysis of collections of (sequential) data definitions and reference patterns to determine constraints that can be placed on data values at various points of executing the source program.

Data Flow Anomaly. A sequence of the events reference (*r*), definition (*d*), and use (*u*) of variables in a program that is either erroneous in itself or often symptomatic of an error.

Data Flow Testing. A testing technique which provides a set of successively more stringent path selection criteria that guide the selection of test data to examine the relationships between variable definitions and variable uses.

Debugger. A software tool intended to assist the user in software fault localization and, potentially, fault correction.

Debugging. The process of correcting syntactic and logical faults detected during testing. Debugging shares with testing certain techniques and strategies, but differs in its usual ad hoc application and local scope.

Decision-To-Decision Path. A sequence of nodes on the control graph that starts at the program entry point or at a decision node, terminates with the program exit or a decision node, and has no decision node in between.

Directed Graph. Consists of a set of nodes interconnected with oriented arcs. An arbitrary directed graph (digraph) may have many entry nodes and many exit nodes. A program digraph has only one entry and one exit.

Driver. A software module that invokes and, perhaps, controls and monitors the execution of one or more other software modules.

Dynamic Analysis. The process of evaluating a system or component based on its behavior during execution.

Emulator. A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

Entry Point. A point in a software module at which execution of the module can begin.

Equivalence Class Partitioning. A test data selection technique based on consideration of partitioning the input domain of a program into a finite number of equivalence classes such that (1) a test of a representative value of each class is equivalent to a test of any other value and (2) each test case should invoke as many different input conditions as possible in order to minimize the total number of test cases necessary.

Error. (1) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. (2) A mental mistake made by a programmer which may result in a program fault.

Error Guessing. A test data selection technique. The selection criteria is to pick values that seem likely to cause failures.

Error Seeding. The process of intentionally adding known faults to those already in a computer program for the purpose of monitoring the rate of detection and removal, and estimating the number of faults remaining in the program.

Essential Knots. A measure of unstructuredness based on control flow knots.

Essential Paths. Program paths that must be executed to achieve 100% coverage.

Exception. An event that causes suspension of normal program execution.

Executable Specification. A specification which is given in a sufficiently formal notation to allow its execution by a computer.

Executable Statement. A statement in a module which is executable in the sense that it produces object code instructions.

Execution Time. (1) The amount of actual or central processor time used in executing a program. (2) The period of time during which a program is executing.

Failure. The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered.

Failure Intensity. Failures per unit of time, the derivative with respect to time of the mean value function of failures.

Failure Intensity Decay Parameter. In the logarithmic Poisson execution time model, the parameter that represents the rate of exponential decay of the failure intensity as a function of mean failures experienced.

Failure Severity. Classification of a failure by its operational impact.

Fault. A manifestation of an error in software. A fault, if encountered, may cause a failure.

Fault Tree Analysis. A form of safety analysis that assesses hardware safety to provide failure statistics and sensitivity analyses which indicate the possible effect of critical failures.

Flowchart. A control flow diagram in which suitably annotated geometrical figures are used to represent operations, data, or equipment, and arrows are used to indicate the sequential flow from one to another.

Formal Specification. In proof of correctness, a description in a formal language of the externally visible behavior of a system or system component. Generally, a specification written and approved in accordance with established standards.

Formal Verification. See Verification.

Function. (1) A specific purpose of an entity or its characteristic action. (2) A subprogram that is invoked during the evaluation of an expression in which its name appears and that returns a value to the point of invocation.

Functional Specification. A set of behavioral and performance requirements which, in aggregate, determine the functional properties of a software system.

Function Points. Function points measure software by quantifying the functionality provided external to itself. It is based primarily on logical design.

Functional Testing. Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

Generic Component. A generic component is one which can be instantiated in a number of predefined ways so that each occurrence of the component can be tailored to suit a particular usage. For example, a generic component which provides a set of queue handling routines might be designed so that it can be instantiated to operate on queues with different message formats.

Global Assertion. Those assertions which are valid for the whole program being validated.

Graph. See **Directed Graph.**

Incident. During testing, any event that occurs during the execution of a software test that requires investigation.

Incremental Analysis. Occurs when (partial) analysis may be performed on an incomplete product to allow early feedback on the development of that product.

Incremental Development. A software development technique in which requirements definition, design, implementation, and testing occur in an overlapping, iterative manner, resulting in an incremental completion of the overall software product.

Independent Verification and Validation (IV&V). Verification and validation of a software product by an organization that is both technically, managerially, and financially separate from the organization responsible for developing the product. See **Validation** and **Verification.**

Infeasible Path. A sequence of program statements that can never be executed.

Information Flow Analysis. A study of the interdependencies of program variables. A given variable *A* will depend on another variable *B* at a specific point in the program if the path taken in reaching that point is such that the value of *A* depends on the value of *B*.

Inspection. A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems. Types include code inspections and design inspections.

Instruction Block. A sequence of statements where execution of the first statement necessarily leads to execution of the last statement.

Instrumentation. Devices or instructions installed or inserted into hardware or software to monitor the operation of a system or component.

Integration. The process of combining software elements, hardware elements, or both into an overall system.

Integration Testing. Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them.

Intervals. Derived from a directed graph, an interval is defined as the following: An interval with head node H is the subgraph containing H plus any nodes that can be reached on a path from H, and which have all their immediate predecessors in the interval. First-order intervals give a count of the intervals that partition the graph into a set of disjoint components. The maximum order is the number of interval iterations required to reduce a graph to a single node.

Invocation. The transfer of control to an entity causing it to be activated.

Linear Code Sequence and Jump (LCSAJ) Program Units. Sections of the code through which the flow of control proceeds sequentially until terminated by a jump in the control flow.

Logarithmic Poisson Execution Time Model. A software reliability model in which the failure process is assumed to be a nonhomogeneous Poisson process with exponentially decreasing failure intensity.

Maintainability. (1) The probability that specified unavailable functions can be repaired or restored to their operational state in the system's intended maintenance environment during a specified period of time. (2) The average effort to locate and fix a software failure.

Metric. A quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Module. A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.

Node. In a diagram, a point, circle, or other geometric figure used to represent a state, event, or other item of interest.

Operational Testing. Testing performed by the end user on software in its normal operating environment.

Parse. To determine the syntactic structure of a language unit by decomposing it into more elementary subunits and establishing the relationships among the subunits. For example, to decompose blocks into statements, statements into expressions, expressions into operators and operands.

Path. In software engineering, a sequence of instructions that may be performed in the execution of a computer program.

Path Analysis. Analysis of a computer program to identify all possible paths through the program, to detect incomplete paths, or to discover portions of the program that are not on any path.

Path Testing. Testing designed to execute all or selected paths through a computer program. (Often paths through the program are grouped into a finite set of classes: one path from each class is tested.)

Performance. The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.

Portability. The ease with which a system or component can be transferred from one hardware or software environment to another.

Pretty Printing. The use of indentation, blank lines, and other visual clues to show the logical structure of a program.

Program Correctness. (1) The extent to which software is free from design defects and coding defects; that is, fault free. (2) Extent to which the software satisfies its specifications and fulfills the user's mission objects. (3) If for all initial states that belong to the set of legitimate initial states, the program *P* terminates with a final state that belongs to the set of legitimate final states, then program *P* exhibits program correctness.

Prototype. A limited implementation of a system built in order to capture or validate some aspects of a system design. The fundamental concept is that a prototype of a system is more

cheaply or more quickly constructed than the actual system. Hence, some aspects of function or performance are typically sacrificed.

Pseudo Code. A combination of programming language constructs and natural language used to express a computer program design.

Quality. (1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations.

Quality Assurance. (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured.

Random Testing. An essentially black-box testing approach in which a program is tested by randomly choosing a subset of all possible input values. The distribution may be arbitrary or may attempt to accurately reflect the distribution of inputs in the application environment.

Regression Testing: Selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, and verify that a modified system or system component still meets its specified requirements.

Reliability. The ability of a system or component to perform its required functions under stated conditions for a specified period of time.

Reliability Model. A model used for predicting, estimating, or assessing reliability.

Reliability Growth. The improvement in reliability that results from correction of faults.

Requirement. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component.

Requirements Specification. A document that specifies the requirements for a system or component. Typically included are functional requirements, performance requirements, interface requirements, design requirements, and development standards.

Retesting. See **Regression Testing.**

Safety. The extent to which the program is protected from causing a specified set of hazards.

Scope. The range within which an identified unit displays itself. Scope of activity refers to the boundaries within which a data structure or program element remains an integral unit. Scope of control refers to the submodules in a program that potentially may execute if control is given to a cited module. Scope of error denotes the set of submodules that are potentially affected by the detection of a fault in a cited module.

Segment. A (logical) segment, or decision-to-decision path, is the set of statements in a module which are executed as a result of the evaluation of some predicate within the module. It begins at an entry or decision statement and ends at a decision statement or exit, and should be thought of as including the sensing of the outcome of a conditional operation and the subsequent statement execution up to and including the computation of the next predicate value, but not including its evaluation.

Self-Checking Software. Software which makes an explicit attempt to determine its own correctness and to proceed accordingly.

Simulation. (1) A model that behaves or operates like a system when provided a set of controlled inputs. (2) The process of developing or using a model as in (1).

Sizing. The process of estimating the amount of computer storage or number of source lines required for a software system or component.

Software. Computer programs, procedures, rules, and any associated documentation pertaining to the operation of a computer system.

Software Fault Tree Analysis. A form of fault tree analysis used for analyzing the safety of software designs or code.

Software Quality. (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, conform to specifications. (2) The

degree to which software possesses a desired combination of attributes. (3) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

Software Reliability. (1) The probability that software will not cause the failure of the system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. (2) The ability of a program to perform a required function under stated conditions for a stated period of time.

Software Reliability Model. A model used for predicting, estimating, or assessing software reliability.

Software Science. Software Science measures the complexity of a software module by calculations based on the incidence of references to operators and operands. The fundamental measures calculated are program vocabulary, length, and volume.

Specification. A document that prescribes in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or system component, and, often, the procedures for determining whether these provisions have been satisfied.

Specification Language. A language, often a machine-processable combination of natural and formal language, used to specify the requirements, design, behavior, or other characteristics of a system or system component.

Statement Testing. Testing designed to execute each statement in a computer program.

Static Analyzer. A software tool that aids in the evaluation of a computer program without executing the program. Examples include syntax checkers, compilers, cross-reference generators, standards enforcers, and flowcharters.

Stress Testing. Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements.

Structural Testing. Testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, and statement testing.

Structured Programming. A well-defined software development technique that incorporates top-down design and implementation and strict use of structured program control constructs.

Stub. A skeletal or special-purpose implementation of a software module, used to develop or test a module that calls or is otherwise dependent upon it.

Symbolic Evaluation. See **Symbolic Execution**.

Symbolic Execution. A software analysis technique in which program execution is simulated using symbols, such as variable names, rather than actual values for input data, and mathematical expressions involving these symbols.

System Testing. Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

Test. A unit test of a single module consists of (1) a collection of settings for the input space of the module, and (2) exactly one invocation of the module. A unit test may or may not include the effect of other modules which are invoked by the module undergoing testing.

Testbed. An environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.

Test Case. A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirements.

Test Data. See **Test Case**.

Test Data Generator. A software tool that accepts as input source code, test criteria, specifications, or data structure definitions; uses these inputs to generate test input data; and, sometimes, determines the expected results.

Test Driver. A program that directs the execution of another program against a collection of test data sets. Usually the test driver also records and organizes the output generated as the tests are run.

Test Management. Management procedures designed to control in an ordered way a large and evolving amount of information on system features to be tested, on system implementation plans, and on test results.

Test Path. The specific (sequence) set of segments that is traversed as the result of a unit test operation on a set of test data. A module can have many test paths.

Test Plan. A document prescribing the approach to be taken for intended testing activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency planning.

Test Repeatability. An attribute of a test indicating whether the same results are produced each time the test is conducted.

Test Log. A chronological record of all relevant details about the execution of a test.

Testability. (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether these criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

Testing. The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

Timing Analyzer. A software tool that estimates or measures the execution time of a computer program or portion of a computer program, either by summing up the execution times of the instructions along the specified paths or by inserting probes at specified points in the program and measuring the execution time between probes.

Top-Down Testing. A systematic testing philosophy which seeks to first test those modules at the top of the invocation structure.

Trace. A record of the execution of a computer program, showing the sequences of instructions executed, the names and values of variables, or both.

Traceability. The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.

Unit. A separately testable element specified in the design of a computer software component.

Unit Testing. Testing of individual hardware or software units or groups of related units. See also **Integration Testing** and **System Testing.**

Unreachability. A statement (or segment) is unreachable if there is no logically obtainable set of input-space settings which can cause the statement (or segment) to be traversed.

Validation. The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Verification. (1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of correctness.

END