NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE TESTING TOOLS:
METRICS FOR MEASUREMENT OF EFFECTIVENESS ON PROCEDURAL AND OBJECT-ORIENTED SOURCE CODE

by

Bernard J. Bossuyt
Byron B. Snyder

September 2001

Thesis Advisor:	J. Bret Michael
Second Reader:	Richard H. Riehle

Report Documentation Page

Report Date 30 Sep 2001	**Report Type** N/A	**Dates Covered (from... to)** -

Title and Subtitle Software Testing Tools: Analyses of Effectiveness on Procedural and Object-Oriented Source Code	**Contract Number**
	Grant Number
	Program Element Number
Author(s) Bernard J. Bossuyt & Byron B. Snyder	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) Research Office Naval Postgraduate School Monterey, Ca 93943-5138	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes	
Abstract	
Subject Terms	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** UU
Number of Pages 209	

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2001	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) Software Testing Tools: Analyses of Effectiveness on Procedural and Object-Oriented Source Code			**5. FUNDING NUMBERS**
6. AUTHOR(S) Bernard J. Bossuyt & Byron B. Snyder			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The levels of quality, maintainability, testability, and stability of software can be improved and measured through the use of automated testing tools throughout the software development process. Automated testing tools assist software engineers to gauge the quality of software by automating the mechanical aspects of the software-testing task. Automated testing tools vary in their underlying approach, quality, and ease-of-use, among other characteristics. Evaluating available tools and selecting the most appropriate suite of tools can be a difficult and time-consuming process. In this thesis, we propose a suite of objective metrics for measuring tool characteristics, as an aide in systematically evaluating and selecting automated testing tools. Future work includes further research into the validity and utility of this suite of metrics, conducting similar research using a larger software project, and incorporating a larger set of tools into similar research.

14. SUBJECT TERMS software testing tool metrics, procedural, object-oriented, software testing tools, metrics, testing tool evaluation, testing tool selection,			**15. NUMBER OF PAGES** 209
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

SOFTWARE TESTING TOOLS: ANALYSES OF EFFECTIVENESS ON PROCEDURAL AND OBJECT-ORIENTED SOURCE CODE

Bernard J. Bossuyt
Lieutenant, United States Navy
B.A., University of Colorado, 1993

Byron B. Snyder
Lieutenant, United States Navy
B.S., United States Naval Academy, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2001

Authors: [illegible]
Bernard J. Bossuyt

Byron B Snyder
Byron B. Snyder

Approved by: James Bret Michael
J. Bret Michael, Thesis Advisor

[illegible]
Richard Riehle, Second Reader

[illegible]
Christopher S. Eagle, Chairman
Computer Science Department

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The levels of quality, maintainability, testability, and stability of software can be improved and measured through the use of automated testing tools throughout the software development process. Automated testing tools assist software engineers to gauge the quality of software by automating the mechanical aspects of the software-testing task. Automated testing tools vary in their underlying approach, quality, and ease-of-use, among other characteristics. Evaluating available tools and selecting the most appropriate suite of tools can be a difficult and time-consuming process. In this thesis, we propose a suite of objective metrics for measuring tool characteristics, as an aide in systematically evaluating and selecting automated testing tools. Future work includes further research into the validity and utility of this suite of metrics, conducting similar research using a larger software project, and incorporating a larger set of tools into similar research.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Sincere appreciation to

Dr. J. Bret Michael & Mr. Richard Riehle

for their guidance and support in this endeavor.

We appreciate the assistance and product evaluation copies

provided by the following testing tool vendors:

LDRA

Parasoft

Telelogic

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

The life cycle of a software component begins with the conceptualization of an information system, and ends with the retirement of the system. Although there have been great improvements in standardizing the software development process, there has yet to be developed a process which guarantees the creation of error-proof software. Testing can be used to assess the quality of software components. However, testing can require a lot of computations when the software component is tested after each step of the software development process or tested to a high-level of assurance. In addition, testing of a software component can be labor intensive, and thus expensive in terms of human capital (e.g., software engineers, project managers, domain experts).

Automated testing tools assist software engineers to gauge the quality of software by automating the mechanical aspects of the software-testing task. Automated testing tools vary in their underlying approach, quality, and ease-of-use, among other characteristics. In addition, the selection of testing tools needs to be predicated on characteristics of the software component to be tested. But how does a project manager choose the best suite of testing tools for testing a particular software component?

We envision the benefits of this research to the Department of the Navy to be twofold. Firstly, the results of our research can be used by software engineers as a basis for selecting the best type of tool or suite of tools for testing the software system under test. Secondly, managers can apply the metrics in order to monitor and gauge the effectiveness of specific combinations of testing tools for software-development projects funded by the US Department of Defense.

B. RESEARCH ISSUES

The goal of this research is to provide project managers with assistance in selecting tools by developing metrics for evaluating software testing tools, in terms of their functionality, usability, and other select distinguishing characteristics.

1. Identifying Metrics

Automated testing tools vary in their ability to both detect known software defects and convey information about these defects to the user of the tool. We developed a list of metrics required to compare testing tools applied to both procedural and object-oriented software.

2. Testing of Procedural versus Object-oriented Source Code

Similar to other software development tools, the focus of some testing tools is on testing procedural software while other tools are tailored for testing object-oriented software. Through our experiments, we have determined that the set of metrics used for comparing tools for use in testing procedural software cannot be one-to-one mapped to those for testing object-oriented software, although the two sets are not disjoint.

3. Evaluating Tools

Through the experimentation with different tools, we have identified metrics that may be used when selecting a tool for a development project. This will assist future efforts in evaluating testing tools' individual strengths and weaknesses and how they relate to the requirements of the software being developed (e.g., procedural vs. object-oriented, vital vs. important).

C. CASE STUDY: CSMA/CD LAN DISCRETE-EVENT SIMULATION PROGRAM

Carrier Sense Multiple Access with Collision Detection (CSMA/CD) is a widely used contention-based access protocol in networks. Prior to transmitting, a station on the network samples the network at its point of connection. If it determines the network is busy, the station delays it transmission. However, if it does not detect any activity it may begin transmitting.

Although a station will not transmit when it detects activity, it is still possible for two station's transmissions to collide. Such a situation occurs when one station begins to transmit prior to another station's transmission reaching the first station's connection point. After transmitting its information, the station monitors the network. If a different packet of information is detected on the network before its transmission has had enough time to reach every station on the network, the station assumes its traffic did not reach its destination.

When a collision is detected, all transmitting stations terminate their activity. The station that first identifies the collision sends a jamming signal to all stations signaling them to cease any transmissions. All stations then wait for the jamming signal to end plus a previously defined period of silence. At that point, any station may begin transmitting.

At the end of the silent period, the previously transmitting stations, and possibly others that now have traffic to send, would detect no activity on the network and therefore would immediately attempt transmission. This would result in another collision. Continuing this cycle would prevent any further traffic on the network.

The description of the protocol to this point, in which a station will transmit with a probability of 1 upon detecting that the bus is available, is classified as non-persistent. To prevent this fatal infinite cycle, *p*-persistent networks were created. In such networks, a station transmits with probability of *p* and delays transmission with probability of $(1 - p)$. Therefore, following a collision, it is not guaranteed that all stations with traffic will attempt to transmit at the end of the silence period. This enables one station's message to be detected by other stations before they retry to transmit, allowing the pending message transmission to take place unimpeded.

The experiments conducted were performed on three versions of programs that simulate a CSMA/CD network. The first version is a procedural program developed by Sadiku and Ilyas with the modification of one line so that it could be operated on a wide range of C and C++ compilers.[1] This version will be referred to as the procedural version. Appendix F contains the flow chart of the simulation program.

This program was selected for this project for two purposes. First, it uses several blocks of code numerous times throughout the program. This factor lends the program to implementation through the use of functions in place of those blocks of code as was done in the second version of the program, hereafter called the functional version. Second, it

[1] Sadiku, M. and Ilyas, M., *Simulation of Local Area Networks,* Boca Raton, Florida. CRC Press, 1994, pp. 112-133.

simulates the interaction of several real-world items that lend themselves to being represented by classes and objects. This approach to simulating the network was used in the third version of the program, which we refer to as the object-oriented version of the program.

II. RELATED WORK

A. IEEE STANDARD 1175 WORKING GROUP'S TOOL-EVALUATION SYSTEM

In December of 1991, a working group of software developers and experienced tool users completed the Reference Model for Computing System-Tool Interconnections, IEEE Standard 1175.[2] As an offshoot of their work, they also introduced a tool-evaluation system. The system implements a set of forms which aides project managers in gathering, organizing, and analyzing information on testing (and other) tools efficiently and, if done correctly, effectively.

The system enables tool evaluators to record tool information in such a way to provide an extensive picture of the tools being considered. The forms allow the evaluators to access tool-dependent factors such as speed, user friendliness, and reliability. They also allow evaluators to access environment-dependent factors such as the cost of the tool, the tool's affect on organizational policies and procedures, and tool interaction with existing organization hardware and software assets. The data forms also facilitate the weighting, rating, and summarizing selection criteria.

Using the forms, project managers have a systematic and repeatable process to follow in selecting tools. The forms assist in developing a list of information needed to select a tool and provide a means to collect, organize, and analyze that information. They also enable evaluators to identify and prioritize user needs, to find what tools are available and most importantly, to select a tool based on estimated cost-effectiveness. The process is performed in five steps: analyzing user needs, establishing selection criteria, tool search, tool selection, and reevaluation.

1. Analyzing User Needs

Just as a full assessment of the customer's needs must be accomplished before beginning a software development process, the first step of the tool selection process is to accurately and completely identify the needs of the prospective tool users. A comprehensive and correct needs analysis is vital to the tool selection process. All future

[2] Poston, Robert R. and Sexton, Michael P., "Evaluating and Selecting Testing Tools", *IEEE Software*, May 1992, pp. 33-42.

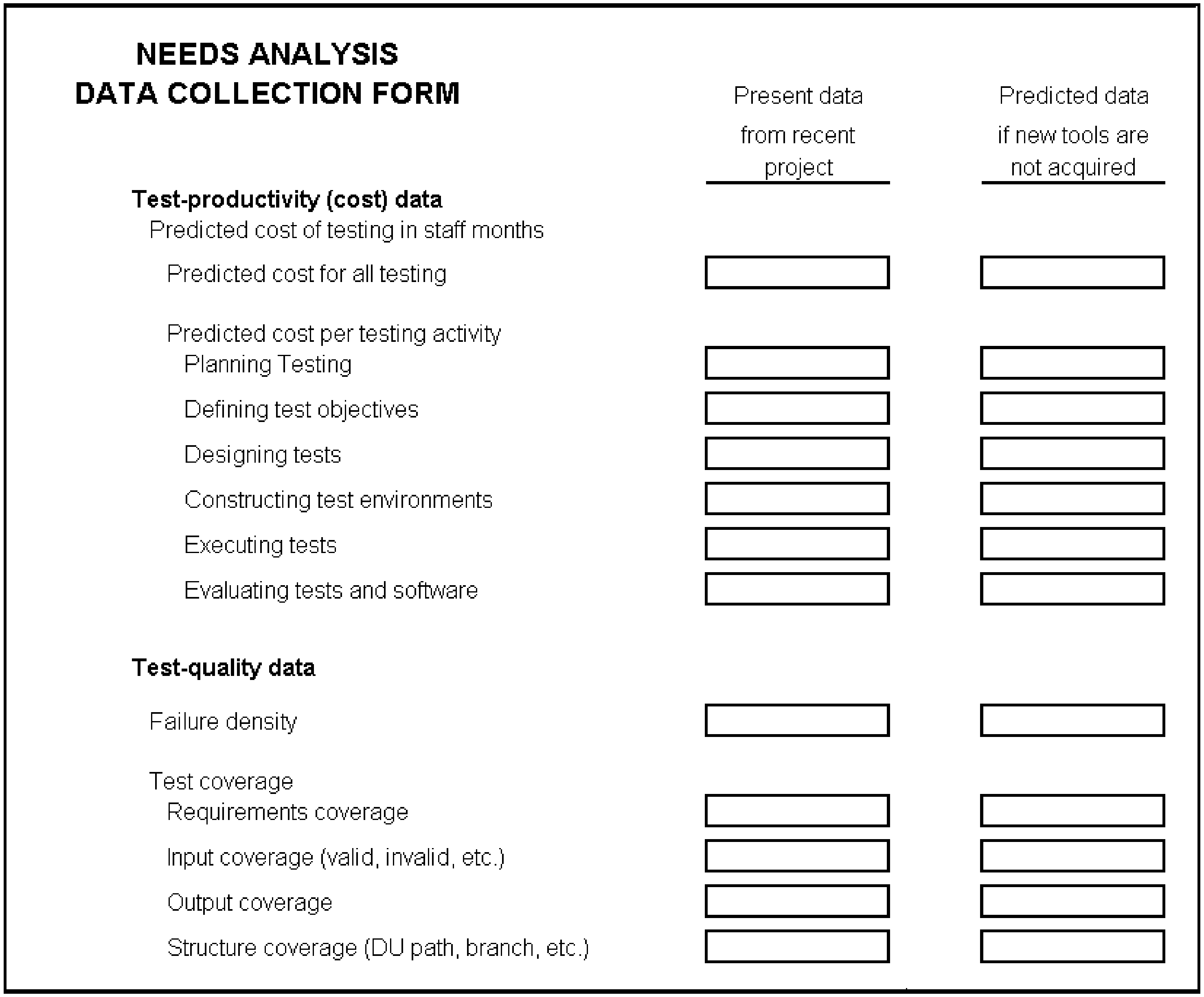
decisions can be traced back to the results of the analysis. Additionally, the effectiveness of the tool selected will be measured against the users' needs that are presented in the analysis, as well as those needs that the analysis overlooked.

The tool-selection process begins the needs analysis step with conversations with the organization's personnel responsible for tracking the company's performance. Speaking with such individuals provides the evaluator with a foundation by identifying the organization's current levels of productivity and quality. Additionally, some organizations also have even higher long-range productivity-level goals. These productivity and quality levels may enable the evaluator to readily eliminate some tools that are not applicable for use in satisfying the organization's goals.

Identifying the amount the organization invests in testing is the next step of the needs analysis process. By combining the current level of testing investment and the number, size and complexity of planned projects, the evaluator can estimate the amount of investment the organization will likely be making on testing in the future. This will aid the evaluator in predicting the likelihood that an investment in a testing tool will provide a positive return on the investment.

Estimating the organization's future testing performance levels is the third step of the needs analysis. Using the data provided by the quality assurance personnel in step one, the evaluator can make educated predictions on future performance. These are generally better than current levels due to the experience and expertise gained in current projects. Additionally, the evaluator must confer with organizational staff to identify any planned adaptations to the current process that may affect testing effectiveness such as hiring of more personnel or changes in procedures. The information gathered thus far is placed in the needs analysis form shown in Figure 1.

The final step of the needs analysis process is the actual analysis of the information mentioned above. Analysis of the information will reveal the expected cost of tool implementation and estimated return on the investment. The organization must weigh the expected improvement in productivity and quality versus the large financial and temporal investment required to integrate an automated testing tool into the



NEEDS ANALYSIS
DATA COLLECTION FORM

	Present data from recent project	Predicted data if new tools are not acquired
Test-productivity (cost) data		
Predicted cost of testing in staff months		
Predicted cost for all testing		
Predicted cost per testing activity		
Planning Testing		
Defining test objectives		
Designing tests		
Constructing test environments		
Executing tests		
Evaluating tests and software		
Test-quality data		
Failure density		
Test coverage		
Requirements coverage		
Input coverage (valid, invalid, etc.)		
Output coverage		
Structure coverage (DU path, branch, etc.)		

Figure 1: Needs Analysis Data Collection Form (From Poston)

organizational procedures. If the organization expects increased projects placing much greater load on the current testing assets with expectations of stable or increased productivity and quality, the level and type of investment need to be further evaluated.

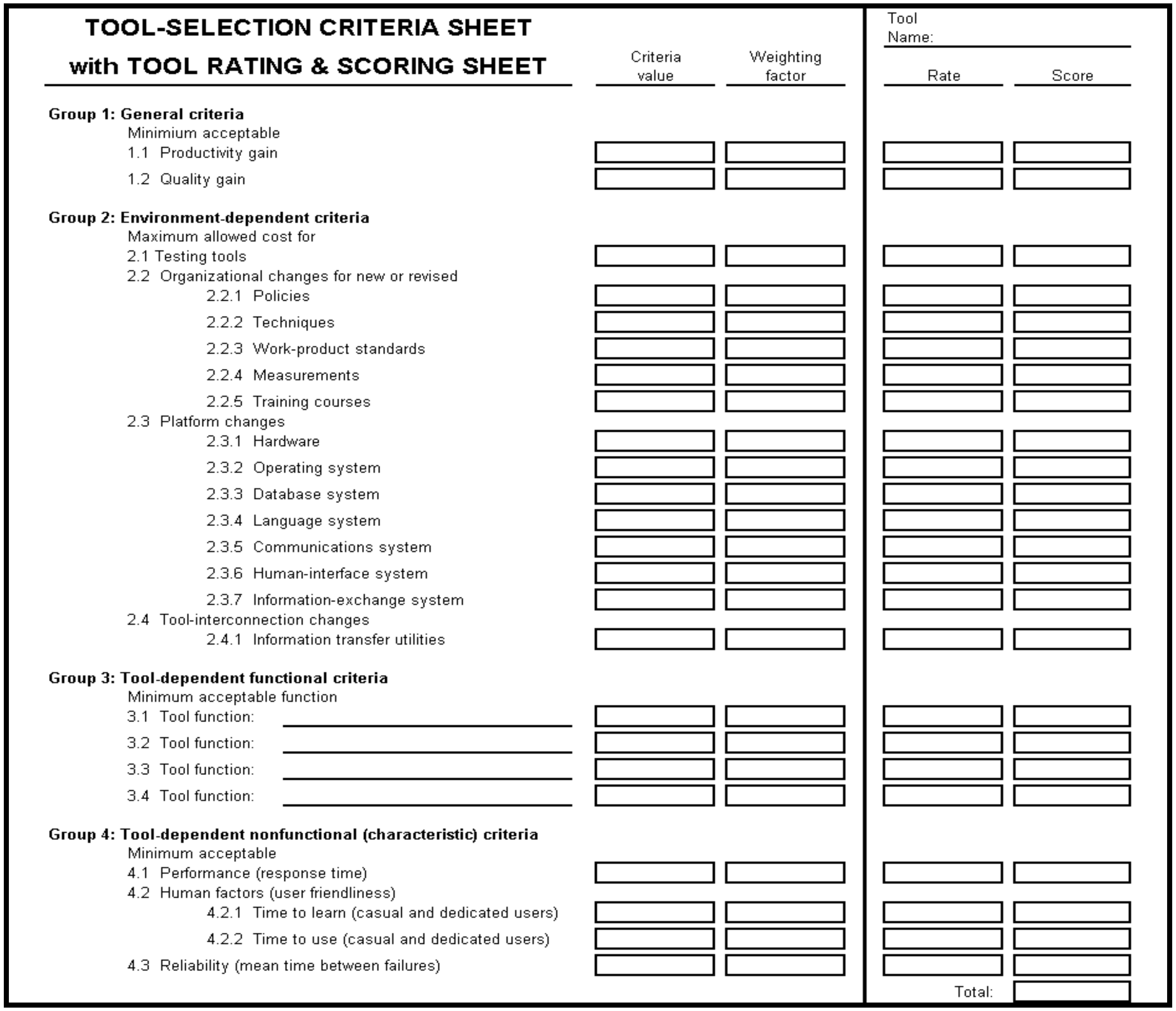
2. Establishing Selection Criteria

The second step of the tool selection process is to establish the criteria that will be used to select the testing tool. The working group has developed a tool-selection criteria form (Figure 2) that organizes several criteria in four groups. The first group is "general criteria" that lists minimum acceptable productivity gain and quality gain. This group is first because if a tool is not expected to provide the required amount of overall productivity and quality improvement, then there is no need to further evaluate the other criteria.

The second group is "environment-dependent criteria." In this group the evaluator determines the maximum amount the organization is willing to spend on tools, organizational changes, platform changes, and tool-interconnection changes. Organizational changes include the costs incurred to make any necessary changes to the organization's policies, techniques, standards, measurements, and training schedules. Platform changes refer to the cost of making adaptations to existing hardware, operating systems, software, and networking assets among others. Tool-interconnection changes are the modifications that must be performed to ensure the data-exchange utilities can continue to perform with the new tool.

If a large number of the expenditure amounts are low, the evaluator may decide to report that the amounts given suggest either a lack of support for acquiring testing tools or a lack of knowledge of the far-reaching implications of attempting to integrate a testing tool into established practice. If the evaluator is unable to elicit enough investment in advance, the assessment may continue. After the selection process has been performed based on the other criteria, the remaining tools can be assessed according to how they would affect each of the investment areas. Then the evaluator may make a recommendation based on the tool which best suits the organization's priorities.

If sufficient data can be collected and they indicate ample support for testing tools, the criteria are then weighed. The weighting process brings together the



TOOL-SELECTION CRITERIA SHEET
with TOOL RATING & SCORING SHEET

Tool Name: ____________________

	Criteria value	Weighting factor	Rate	Score
Group 1: General criteria				
Minimium acceptable				
1.1 Productivity gain				
1.2 Quality gain				
Group 2: Environment-dependent criteria				
Maximum allowed cost for				
2.1 Testing tools				
2.2 Organizational changes for new or revised				
2.2.1 Policies				
2.2.2 Techniques				
2.2.3 Work-product standards				
2.2.4 Measurements				
2.2.5 Training courses				
2.3 Platform changes				
2.3.1 Hardware				
2.3.2 Operating system				
2.3.3 Database system				
2.3.4 Language system				
2.3.5 Communications system				
2.3.6 Human-interface system				
2.3.7 Information-exchange system				
2.4 Tool-interconnection changes				
2.4.1 Information transfer utilities				
Group 3: Tool-dependent functional criteria				
Minimum acceptable function				
3.1 Tool function: ______				
3.2 Tool function: ______				
3.3 Tool function: ______				
3.4 Tool function: ______				
Group 4: Tool-dependent nonfunctional (characteristic) criteria				
Minimum acceptable				
4.1 Performance (response time)				
4.2 Human factors (user friendliness)				
4.2.1 Time to learn (casual and dedicated users)				
4.2.2 Time to use (casual and dedicated users)				
4.3 Reliability (mean time between failures)				
			Total:	

Figure 2. Tool Selection Criteria Sheet
with Tool Rating and Scoring Sheet (From Poston)

appropriate entities, which assign ranking to each criterion based on its perceived or actual importance. Each item on the list must have a unique value. The process is useful in moving the organization closer to a consensus on those requirements that are the most important and empowers the users of tools and tool-generated information to provide input to decision-makers.

3. Tool Search

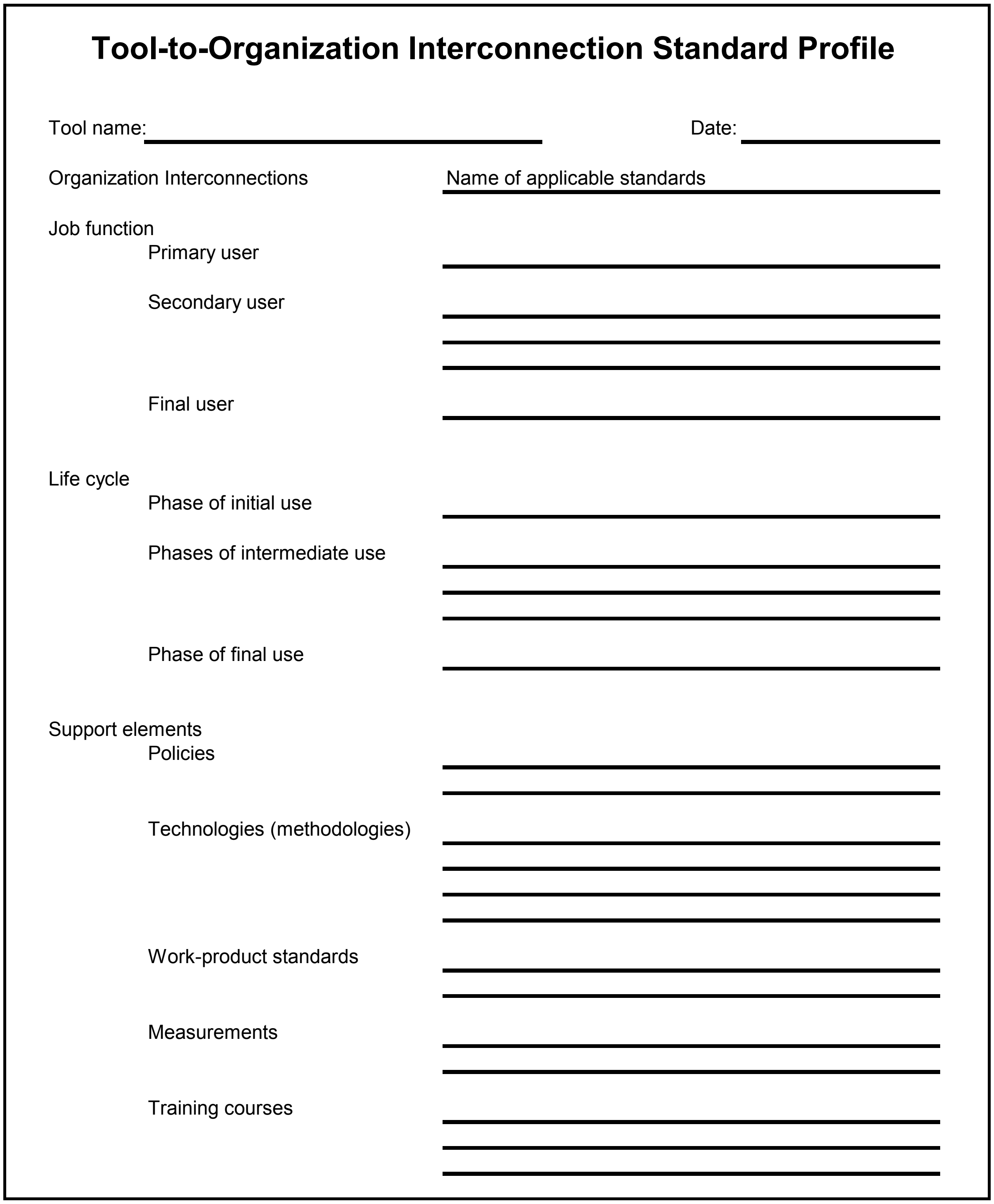
Searching for available tools is the third step in the tool-selection process. The actual search begins after the evaluator has developed an organizational profile. By changing 'tool name' to 'organization name' on the "Tool-to-Organization," "Tool-to-Platform," and "Tool-Interconnection" profile forms of IEEE 1175 (Figures 3-5), they are converted to organizational profile forms. These forms are then completed to the extent the organization deems to be necessary, while documenting any deviation from the standard.

Once the profile of the organization is complete, it is used to eliminate tools from the vast listings of tool surveys that are available from several sources listed in Poston and Sexton. A search of current trade publications, academic journals, and Internet search engines may also provide other prospective tools.

When the evaluator is satisfied with the list of potential tools, a request is made to the suppliers of those tools for the most recent data available on the tool's characteristics. The recommended approach is to provide blank versions of the tool-profile forms from IEEE 1175. Along with the completed forms, the evaluator should request current purchase price, a list of current users who will be willing to discuss their experiences with the tool, and trade articles, research or other independent sources of information that may attest to the tool's value to the testing process. The quality of response may be indicative of the level of support the supplier will provide if the tool is selected. When the evaluator is satisfied with the amount of responses, the tool selection process may begin.

4. Tool Selection

The evaluator is responsible for assessing how closely a tool compares to the criteria developed in the second step (Figure 2). Using all of the information gathered on a tool, the assessor assigns a rating of one if the tool exactly matches the set criteria. If

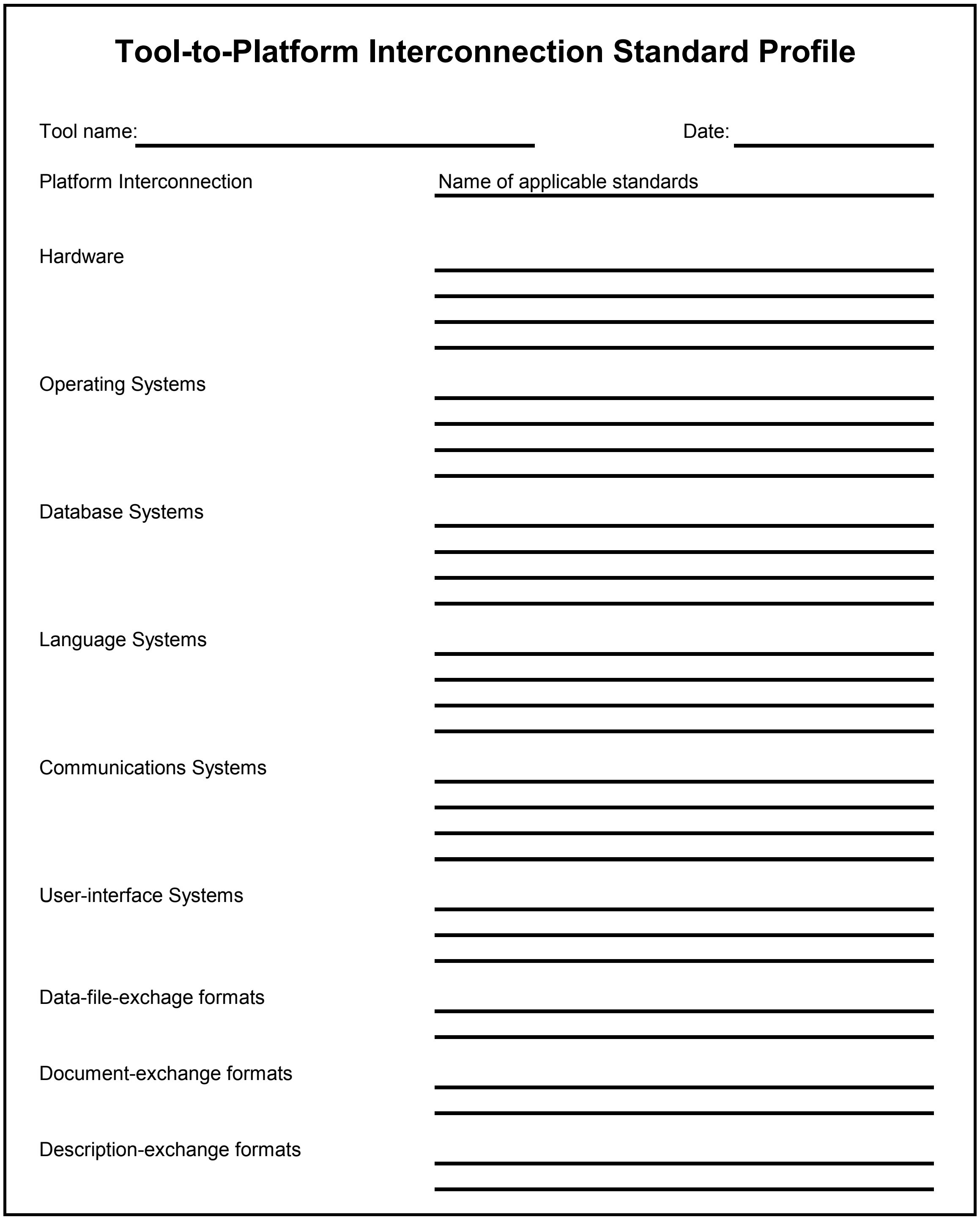


Tool-to-Organization Interconnection Standard Profile

Tool name: ____________________ Date: __________

Organization Interconnections	Name of applicable standards
Job function	
Primary user	
Secondary user	
Final user	
Life cycle	
Phase of initial use	
Phases of intermediate use	
Phase of final use	
Support elements	
Policies	
Technologies (methodologies)	
Work-product standards	
Measurements	
Training courses	

Figure 3. Tool-Organization Form (From Poston)

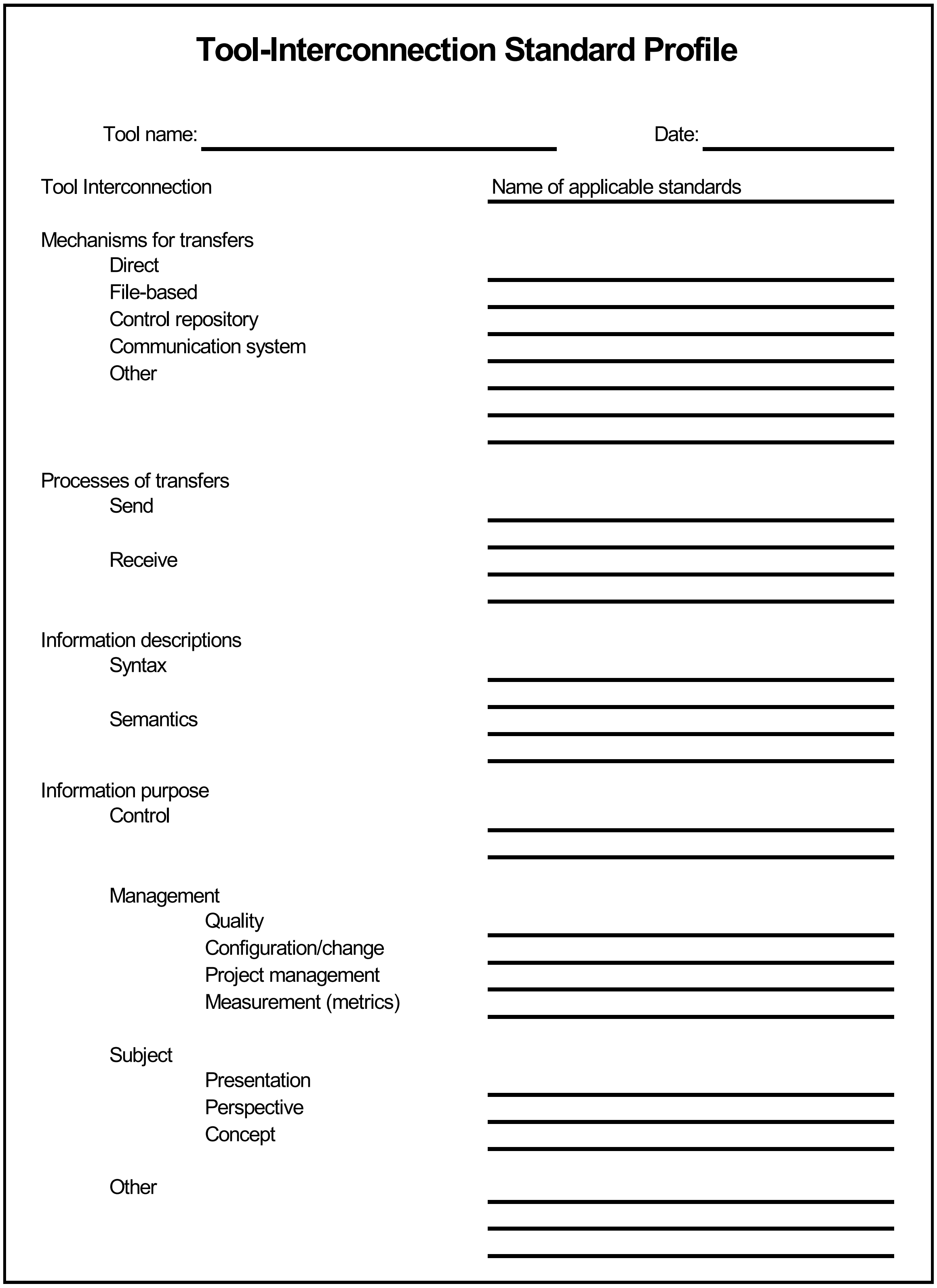


Tool-to-Platform Interconnection Standard Profile

Tool name: ____________________ Date: ____________

Platform Interconnection	Name of applicable standards
Hardware	
Operating Systems	
Database Systems	
Language Systems	
Communications Systems	
User-interface Systems	
Data-file-exchage formats	
Document-exchange formats	
Description-exchange formats	

Figure 4. Tool-Platform Form (From Poston)



Tool-Interconnection Standard Profile

Tool name: ____________________ Date: __________

Tool Interconnection	Name of applicable standards
Mechanisms for transfers	
Direct	
File-based	
Control repository	
Communication system	
Other	
Processes of transfers	
Send	
Receive	
Information descriptions	
Syntax	
Semantics	
Information purpose	
Control	
Management	
Quality	
Configuration/change	
Project management	
Measurement (metrics)	
Subject	
Presentation	
Perspective	
Concept	
Other	

Figure 5. Tool Interconnection Standard Profile Form (From Poston)

the tool does not support the criterion in any way, a rating of zero is assigned. If a tool only partially provides the needed support, a value between zero and one is assigned. For example, if a tool only provides 25% of the desired productivity gain, a value of .25 is placed in the productivity gain rate line. The rating is then multiplied by the weighting given to the criterion, in order to provide the tool's final score for that criterion. Each tool's scores are then tallied.

The tool with the greatest total score is likely the one that the evaluator should recommend for use to the project managers, tool users and other personnel who have input in the final decision about which tool to use. Meeting participants listen to the evaluator's scoring and make assessments about each tool. The group then decides on which tool should be adopted.

5. Reevaluation

After the selected tool has been implemented, it is continually reevaluated. The tool evaluator and project managers work together to compare actual tool performance to what was expected and promised. If the tool fails to perform or meet expectation, then an analysis is conducted to determine whether the shortcomings are inherent to the tool or are problems that the supplier may resolve through product improvements, training, or other means. The cost of corrections will need to be weighed against the expected return as well as the cost of adopting another tool.

6. Summary

The tool selection process developed by the IEEE 1175 Working Group provides a solid foundation upon which to build an organization's tool selection procedures. The forms provide an excellent starting point for ranking prospective tools, but an organization should include other criteria it feels are important and remove those that are perceived of lesser importance to the organization. We build upon this list of criteria and identify separate sets of criteria for tools applied to procedural code and for those applied to object-oriented code.

B. INSTITUTE FOR DEFENSE ANALYSES REPORTS

The Institute for Defense Analyses published *An Examination of Selected Software Testing Tools: 1992*[3] with a follow up supplement a year later[4]. These reports document the results of research conducted on testing tools. While the tools and knowledge gathered may be dated, they were still useful in our research. They provide a historical frame of reference for the recent advances in testing tools as well as identify a large number of measurements that may be used in assessing testing tools. For each tool, the report details different types of analysis conducted, the capabilities within those analysis categories, operating environment requirements, tool-interaction features, along with generic tool information such as price, graphical support, and the number of users.

The research was conducted to provide software developers with information regarding how software testing tools may assist the development and support of software to be used for the Strategic Defense Initiative (SDI). The major conclusions of the study were that:

- Test management tools offered critical support for planning tests and monitoring test progress.
- Problem reporting tools offered support for test management by providing insight software products' status and development progress.
- Available static analysis tools of the time were limited to facilitating program understanding and assessing characteristics of software quality.
- Static analysis tools provided only minimal support for guiding dynamic testing.
- Many needed dynamic analysis capabilities were not commonly available.
- Tools were available that offered considerable support for dynamic testing to increase confidence in correct software operation.

[3] Youngblut, C and Brykczynski B., *An Examination of Selected Software Testing Tools: 1992*, December 1992.

[4] Youngblut, C and Brykczynski B., *An Examination of Selected Software Testing Tools: 1993 Supplement*, October 1993.

- Most importantly, they determined that the wide range of capabilities of the tools and the tools' immaturity required careful analysis prior to selection and adoption of a specific tool.

C. SOFTWARE TECHNOLOGY SUPPORT CENTER'S SOFTWARE TEST TECHNOLOGIES REPORT

The Software Technology Support Center works with Air Force software organizations to identify, evaluate and adopt technologies to improve product quality, increase production efficiency, and hone cost and schedule prediction ability.[5] Section four of their report discusses several issues that should be addressed when evaluating testing tools and provides a sample tool-scoring matrix. Current product critiques and tool evaluation metrics and other information can be obtained by contacting them through their website at http://www.stsc.hill.af.mil/SWTesting/.

[5] Daich, Gregory T., etal, *Software Test Technologies Report*, August 1994, p. 1.

III. METHODOLOGY

A. TOOL SEARCH

The following is a brief summary of testing tools that we considered using in our thesis research. These tools were selected based on whether or not they support C++ and also whether or not they could be run on a Windows platform.

1. BoundsChecker

a. Summary

BoundsChecker is a Compuware Numega product that automatically detects static, stack and heap memory errors and resource leaks. The product assists in finding and fixing memory and resource leaks and API, pointer, and memory errors automatically. BoundsChecker identifies the line of source code where errors occurred, provides explanations, and provides suggested solutions and coding samples.

b. Features

- Detects memory and resource leaks
- Finds and fixes failed API calls in any application, component, DLL or EXE
- Identifies static, stack and heap memory errors
- Identifies the exact line of code where an error occurs
- Verifies that your code will run properly on all Win32 platforms, including Windows CE
- Works from within the Visual C++ Developer Studio environment

2. C-Cover

a. Summary

C-Cover is a full-featured code coverage analyzer for C/C++ running on Microsoft and Unix systems offered by Bullseye Testing Technology. C-Cover finds untested code and measures testing completeness. Test productivity is increased and time is saved by identifying untested control structures.

b. Features

- Ability to include or exclude any portion of project code

- Automatic merging of multiple sessions
- Automatic merging of coverage for DLLs and shared libraries that are used by multiple programs
- Backed by premier level technical support
- Five levels of detail: source directory, source file, C++ class, function, and control structure
- Full support for both C++ and C including templates, exception handling, inline functions, namespace
- Function coverage and condition/decision coverage measurement
- Graphical Coverage Browser for Windows
- Indexed HTML user documentation
- Many options for searching, filtering, and sorting report information
- Run-time source code included
- Sample programs and Visual Studio projects
- Simple floating license
- Support for DLLs, shared libraries, device drivers, ActiveX, DirectX, COM, and time-critical applications
- Support for languages translated to C/C++, such as lex and yacc
- Support for multiple threads, processes, users
- Transparent operation

3. CTC++ (Test Coverage Analyzer for C/C++)

a. Summary

CTC++, a Testwell Oy product, is an instrumentation-based tool supporting testing and tuning of programs written in C and C++ programming languages. CTC++ measures test coverage and reports on the dynamic behavior of the program under test.

b. Features

- Ease of use
- Independent instrumentation of source files
- Integrated to Microsoft Visual C++ Developer Studio

- Support for host-target and kernel code testing
- Usable with GUI testers

4. Cantata++

a. Summary

Cantata++ supports unit and integration testing of C++ software. Cantata++ offers dynamic testing and test coverage analysis of C++ applications. Cantata++ measures object-oriented coverage measures such as inheritance and instantiations of templates. This tool offers minimal intrusion into the development and testing process by not generating stubs. Cantata++ is a product of Quality Checked Software.

b. Features

- Access to data within the implementation of the software under test
- Full control of the interface between the software under test and other software
- Data value checking facilities
- Support for reuse of test case
- Test harness for developing structured, repeatable tests
- Testing of exceptions and templates

5. ObjectChecker/Object Coverage/ObjectDetail

a. Summary

These three tools are all products of ObjectSoftware Inc. ObjectChecker helps automate the style of C++ code and compares C++ constructs with pre-defined coding rules. ObjectCoverage helps automate and improve the effectiveness of software testing by analyzing “selection” and “iteration” statements and generating a test case/branch coverage report. ObjectDetail is a tool to locate early defects and build software-oriented metrics for analysis of an application.

b. Features

- Allows the use of regular expressions to define rules for the style checker, to suppress report generation for selected files and methods and selected sections of code
- Can be invoked from within test scripts
- Generates violation reports in PostScript or regular text format
- Handles all C++ classes (templates and non-templates)
- No source code changes are required
- Works at the source level

6. Panorama C/C++

a. Summary

Panorama C/C++ is a product of ISA (International Software Automation Inc.). Panorama C/C++ is a fully integrated software engineering environment that supports both the development of a new system and the improvement of an existing system. The environment supports software testing, quality assurance, maintenance, and re-engineering.

b. Features

- Automated defect detection with capability to identify the location of the source code segment/branch having a defect found
- Automated error simulation
- Improves test planning through complexity analysis, control flow analysis, and control flow diagramming
- Detailed data analysis, including the analysis of global and static variables to identify where they are defined, used, or assigned a value
- Logic-error checking through program-logic analysis and diagramming
- Program review and inspection
- Test case design through path analysis and automatic extraction of path-execution conditions

- Incremental unit testing by assigning bottom-up unit-test orders without using stubs
- Performance analysis and module/branch execution frequency analysis to locate performance bottlenecks

7. TCAT C/C++

a. Summary

TCAT C/C++ detects weaknesses in code. Easily accessible point-and-click coverage reports find the segments that need to be further tested. Digraphs and call-trees can be viewed pictorially. TCAT C/C++ is offered by Software Research Inc.

b. Features

- Common user interface
- Support for C++ templates, in-line functions and exception handlers
- Support for Microsoft Foundation Classes
- Interfaces to handle large, multiple complex projects
- Automated product installation
- Point-and-click coverage reporting
- Fully integrated with Microsoft Visual C++
- GUI is fully integrated with some C++ compilers

B. TOOLS SELECTED FOR EVALUATION

1. LDRA TESTBED

a. Summary

LDRA Testbed is a source code analysis and test coverage measurement tool. Testbed utilizes its own parsing engine, offering the user more flexibility for tailoring the tool to meet requirements. As a complete package of modules integrated into an automated, software testing toolset, LDRA Testbed enables attaining a greater degree of software testing.

LDRA Testbed's two main testing domains are Static and Dynamic Analysis. Static Analysis analyzes the code, while Dynamic Analysis involves execution with test data to detect defects at run time. LDRA Testbed analyzes the source code, producing reports in textual and graphical form depicting both the quality and structure of the code, and highlighting areas of concern.

LDRA Testbed supports the C, C++, ADA, Cobol, Coral66, Fortran, Pascal, and Algol programming languages. It has been ported to the following operating systems: MS Windows NT/2000/9x/Me, Digital Unix, HP-UX, AIX, SCO ODT, SGI Irix, SunOS 4 (Solaris. 2.1), Solaris Sparc/Intel, VAX/VMS, OpenVMS, MVS, Unisys A Series, and Unisys 2200 Series.

b. Static Analysis Features

Main Static Analysis is the kernel module of the LDRA Testbed system. All software requiring LDRA Testbed analysis must first be processed by Main Static Analysis.

Main Static Analysis produces the following:

- LCSAJ Report (see page 23 for a description of LCSAJ)
- Metrics Report
- Quality Report
- Reformatted Code

1.) Complexity Analysis: Complexity measures can be computed for procedures, files and even across an entire system. Complexity Analysis analyzes the subject code, reporting on its underlying structure on a procedure-by-procedure basis.

2.) Metrics Report: Complexity metrics are reported in the Metrics Report. This configurable report breaks down each metric on either a file-by-file or a procedure-by-procedure basis and stipulates whether the value has passed the quality model or not. At the top of the report is a list of the metrics that are computed.

Each metric is reported with those passing the quality model in green, and those failing in red.

Complexity Metric Production: In order to control the quality of software products, *LDRA Testbed* produces the following complexity metrics:

Control Flow Knots: Knot analysis measures the amount of disjointedness in the code and hence the amount of 'jumping about' a code reader will be required to undertake. Excessive knots may indicate that a program can be reordered to improve its readability and complexity.

Cyclomatic Complexity: Cyclomatic Complexity reflects the decision-making structure of the program. It is recommended that for any given module the metric should not exceed ten. This value is an indicator of modules that may benefit from redesign. It is a measure of the size of the directed graph, and hence is a factor in complexity.

Reachability: All executable statements should be reachable by following a control-flow path from the start of the program. Unreachable code consists of statements for which there is no such path. *LDRA Testbed* marks all such lines as being unreachable. Since they contribute nothing to the program computations, they could be removed without altering the code's current functionality.

Looping Depth: The maximum depth of the control flow loops is a factor in the overall readability, complexity and efficiency of the code.

LCSAJ Density: The LCSAJ density is a maintainability metric. If a line of code is to be changed, then the density informs the user how many test paths (LCSAJs) will be affected by that change. If the density is high, then confidence that the change is correct for all test paths will be reduced, and hence an increased amount of regression testing may be required.

An LCSAJ is a linear sequence of executable code commencing either from the start of the program or from a point to which control flow may jump. It is terminated either by a specific control-flow jump or by the end of the program. The linear code sequence consists of one or more consecutive Basic Blocks. Consequently, there may be a number of predicates that must be satisfied in order for control flow to execute the linear code sequence and terminating jump.

Comments: To control readability and maintainability, the following are measured:

- Number of lines of comments in the declaration part of a procedure
- Number of lines of comments in the executable part of a procedure
- Number of lines of comments just prior to a procedure declaration (a procedure header)
- Number of totally blank lines of comments

Halstead's Metrics: These metrics measure the size of a program. *LDRA Testbed* supports the following Halstead metrics:

- Length
- Total Operands
- Total Operators
- Unique Operands
- Unique Operators
- Vocabulary
- Volume

3.) Quality Report: The Quality Report creates views of the quality of the source code. The report can reflect the quality of a single file, the entire system or a group of unrelated source files, and can be produced in either ASCII or HTML format.

c. Dynamic Analysis Features

After the source code has been instrumented, compiled, and linked, the execution of the program creates an output stream which contains the execution history. The Dynamic Coverage Analysis option processes the execution history, mapping its information onto the control flow information on the source code acquired from the static analysis phase. The instrumented program is normally executed several times, with each execution followed by a Dynamic Coverage Analysis. The remaining options are usually selected after completion of a series of these analyses.

The output of the Dynamic Coverage Analysis can be used as input for making decisions about improving the robustness of the source code. In essence, the dynamic-coverage metrics provide an indication of how much more needs to be done.

To increase the coverage, the user must construct further sets of test data to be run with the instrumented source code. The Dynamic Coverage Analysis must then be rerun. Each run of the Dynamic Coverage Analysis shows the names of the entire test data sets used to obtain the corresponding results. The detailed results are reported as profiles, including the following:

- Untested Code Reporting
- Control Flow Tracing
- Statement execution Frequency Analysis
- Branch Execution Frequency Analysis
- LCSAJ Sub-path Execution Analysis
- Profile Analysis
- Dynamic Data Set Analysis
- Host/Target Testing
- Real-Time Systems Testing

2. Parasoft Testing Products

a. Summary

This evaluation consisted of the following Parasoft Products: C++ Test with embedded CodeWizard (beta version 1.3 August 2, 2001) and Insure++. C++ Test is a C/C++ unit testing tool that automatically tests any C/C++ class, function, or component without requiring the user to develop test cases, harnesses, or stubs. C++ Test automatically performs white-box testing, black-box testing, and regression testing.

CodeWizard can enforce over 170 industry-accepted C/C++ coding standards and permits the user to create custom rules that apply to a particular software-development effort.

Insure++ automatically detects runtime errors in C/C++ programs.

Parasoft's Testing Tool suite supports Microsoft Visual Studio 6.0 on Windows NT/2000. Parasoft is developing Linux and Solaris versions at the time this research was conducted.

b. C++ Test features

- Can be used to achieve and demonstrate various levels of code coverage
- Allows verification of class functionality and construction without waiting for the rest of the system to build
- Automates unit testing process, which is fundamental to Extreme Programming and other programming models
- Automatically builds scaffolding and stubs for classes/functions
- Automatically creates and executes test cases for classes/functions
- Performs automatic regression testing
- Performs component testing
- Provides a framework for entering and executing specification and functionality tests
- Provides for incremental testing of classes/functions
- Tests code under extreme cases

c. CodeWizard features

- Coding standards grouped according to programming concepts and severity
- Enforces predefined and custom rules
- Supports the creation of custom rules
- Supports user-specified thresholds for triggering rule-violation reports
- Supports rule suppression

d. *Insure++ features*

- Allows switching between selective checking and thorough checking with full instrumentation
- Capable of checking third-party libraries, functions, and interfaces to modules written in languages other than C
- Contains a large set of rules for detecting errors specific to C++ code
- Detects numerous categories of errors such as memory corruption, memory leaks, memory allocation errors, variable initialization errors, variable definition conflicts, pointer errors, library errors, logic errors, and algorithmic errors
- Identifies the source and location of leaks
- Supports cross-platform development
- Supports large number of compilers

3. Telelogic Products

a. *Summary*

Logiscope TestChecker measures structural test coverage and shows uncovered source code paths. Logiscope TestChecker is based on a source code instrumentation technique that can be tailored to the test environment.

Logiscope TestChecker identifies which parts of the code remain untested. It also identifies inefficient test cases and regression tests that should be re-executed when a function or file is modified. Logiscope TestChecker is based on source code instrumentation techniques such as the use of probes.

b. *Features*

- Assesses test case efficiency and testing progress
- Assists in the definition of regression tests
- Instrumentation code probes can be tuned to meet application-execution constraints, for
    - Deciding how to implement memory management coverage data (e.g., statically or dynamically, type of data)

    - Reducing overhead associated with the size of the instrumented files
    - Selecting the type of output device for dumping coverage data (e.g., file, TCP/IP, serial link)
- Merges multiple test run results
- Provides code-coverage analysis
- Provides information for each function, file or project:
    - Decision coverage
    - Modified conditions/decisions (MC/DC) as per DO-178B
    - List of tests that cover the selected function or file
- Provides information for each test case:
    - Specific test case improvement regarding overall overage
    - List of impacted files and functions
    - Description field for user comments
- Supports multi-threaded applications
- Tests can be performed either on host or target platforms
- Computes the following metrics:
    - Call-pair coverage
    - Instruction-blocks coverage
- Automatically generates customizable reports and documentation
- Provides a visual representation of the uncovered paths (i.e., call and control-flow graphs)

C. SOFTWARE QUALITY METRICS

Extensive research and numerous writings have been completed in relation to metrics for measuring quality of software programs. The history of software metrics began with counting the number of lines of code (LOC). It was assumed that more lines of code implied more complex programs, that in turn were more likely to have errors.

However, software metrics have evolved well beyond the simple measures introduced in the 1960s.

1. Procedural (Traditional) Software Metrics

Metrics for traditional or procedural source code have increased in number and complexity since the first introduction of lines of code. While LOC is still used, it is rarely measured simply to know the length of procedural programs since there continues to be debate on the correlation between size and complexity. Instead, LOC is used in the computation of other metrics, most notably, in determining the average number of defects per thousand lines of code.

McCabe[6] first applied cyclomatic complexity to computer software. Cyclomatic complexity is an estimate of the reliability, testability, and maintainability of a program, based on measuring the number of linearly independent paths through the program. Cyclomatic complexity is measured by creating a control graph representing the entry points, exit points, decision points, and possible branches of the program being analyzed. The complexity is then determined using the following formula:

Equation 3.1

$M = V(G) = e - n + 2p$ where $V(G)$ is the cyclomatic number of G

e is the number of edges

n is the number of nodes

p is the number of unconnected parts of G

This metric however does not look at the specific implementation of the graph. For example, nested if-then-else statements are treated the same as a case statement even though their complexities are not the same.

Function point (FP)[7] is a metric that may be applied independent of a specific programming language, in fact, it can be determined in the design stage prior to the commencement of writing the program. To determine FP, an Unadjusted Function Point

[6] McCabe, "Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320, December 1976.

[7] Dekkers, C., "Demystifying Function Points: Let's Understand Some Terminology," *IT Metrics Strategies*, October 1998.

Count (UFC) is calculated. UFC is found by counting the number of external inputs (user input), external outputs (program output), external inquiries (interactive inputs requiring a response), external files (inter-system interface), and internal files (system logical master files). Each member of the above five groups is analyzed as having either simple, average or complex complexity, and a weight is associated with that member based upon a table of FP complexity weights. UFC is then calculated via:

Equation 3.2

$$\text{UFC} = \Sigma_{1\text{->}15}\ (\text{number of items of variety i}) \times (\text{weight of i})$$

Next, a Technical Complexity Factor (TCF) is determined by analyzing fourteen contributing factors. Each factor is assigned a score from zero to five based on its criticality to the system being built. The TCF is then found through the equation:

Equation 3.3

$$\text{TCF} = 0.65 + 0.01\Sigma_{1\text{->}14}\ F_i$$

FP is the product of UFC and TCF. FP has been criticized due to its reliance upon subjective ratings and its foundation on early design characteristics that are likely to change as the development process progresses.

Halstead[8] created a metric founded on the number of operators and operands in a program. His software-science metric (also referred to as 'halted length') is based on the enumeration of distinct operators and operands as well as the total number of appearances of operators and operands. With these counts, a system of equations is used to assign values to program level (i.e., program complexity), program difficulty, potential minimum volume of an algorithm, and other measurements.

[8] Halstead, M., *Elements of Software Science*, Elsevier, North-Holland, New York, 1977.

2. Object-Oriented Software Metrics

The most commonly cited object-oriented software metrics are those proposed by Chidamber and Kemmerer.[9] Their suite consists of the following metrics: weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, response for a class, and lack of cohesion in methods.

Weighted methods per class (WMC) is the sum of the individual complexities of the methods within that class. The number of methods and the sum of their complexities correlate to the level of investment of time and effort in designing, developing, testing, and maintaining the class. Additionally, a large number of methods implies increased complexity due to the increased likelihood of their use by children of the class.

Depth of inheritance tree (DIT) is defined as the maximum length from the node to the root of a class tree. The deeper a class is in the inheritance hierarchy, the greater the likelihood that it inherits a large number of methods, thereby making its behavior more complex to both predict and analyze. Also, a larger DIT implies greater design complexity due to the larger number of classes and methods in the project.

The number of immediate subclasses of class is represented by "number of children" (NOC). A larger NOC implies a significant amount of inheritance and reuse. The more times a class is inherited, the greater the possibility that errors will be made in its abstraction and the greater the possible impact the class has on the project. Therefore, a class with a high NOC may need to be tested more thoroughly than classes with lower NOC's.

Coupling between object classes (CBO) is defined as the number of classes to which it is coupled (i.e., interdependent on). When a class inherits methods, instance variables, or other characteristics from another class, they are coupled. The greater the number of shared attributes, the greater the interdependence. A significant amount of coupling leads to an increased probability of changes in one class causing unaccounted, and possibly undesired, changes in the behavior of the other. This tighter coupling may require more extensive testing of classes that are tightly coupled together.

[9] Chidamber, S. and Kemmerer, C., 'A Metrics Suite for Object Oriented Design', *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, pp. 476-493, 1994.

Response for a class (RFC) is defined as the cardinality of the set whose members are the methods of the class that can potentially be called in response to a message received by an object in that class. The set's members include the class methods called by other methods within the class being analyzed. A large RFC indicates that there are numerous ways in which class methods are called, possibly from many different classes. This may lead to difficulties in understanding the class, making analysis, testing, and maintenance of the class uncertain.

Lack of cohesion in methods (LCOM) is defined as the number of method pairs with no shared instance variables minus the number of method pairs with common attributes. If the difference is negative, LCOM is set equal to zero. A large LCOM value indicates strong cohesion within the class. A lack of cohesion, indicated by a low LCOM value, signifies that the class represents two or more concepts. The complexity of the class, and perhaps of the entire project, could be reduced by separating the class into smaller, and likely simpler, classes.

Chidamber and Kemmerer's suite were extended by Lie and Henry.[10] They

introduced the Message Passing Coupling (MPC) metric that counts the number of send statements defined in a class; this signifies the complexity of message passing between classes. Their Data Abstraction Coupling (DAC) metrics is calculated based on the number of abstract data types used in the class and defined in another class. The greater the DAC value, the greater the dependence on other classes and therefore the greater the complexity of the project.

Henry and Kafura developed the Information Flow Complexity (IFC) metric to measure the total level of information flow of a module.[11] A module's (M) fan-in is defined as the number of local flows that terminate at M plus the number of data structures from which information is retrieved by M. Fan-out is defined as the number of local flows that emanate from M plus the number of data structures that are updated by M. Local flow is defined as either a module invoking a second module and passing

[10] Lie, W. and Henry, S., "Object-oriented Metrics that Predict Maintainability", *Journal of Systems and Software,* Vol. 23, No. 2, pp. 111-122.

[11] Henry, S. and Kafura, D., "Software Structure Metrics based on Information Flow," *IEEE Transactions on Software Engineering*, SE Vol. 7 No. 5, September 1981.

information to it or a module being invoked returning a result to the calling module. IFC is then found by summing the LOC of M and the square of the product of M's fan-in and fan-out. Shepperd removed LOC to achieve a metric more directly related to information flow.[12]

Equation 3.4

$$IFC(M) = LOC(M) + [\text{fan-in}(M) \times \text{fan-out}(M)]^2$$

Lorenz and Kidd proposed another set of object-oriented software quality metrics.[13] Their suite includes the following:

- Number of scenarios scripts (use cases) (NSS)
- Number of key classes (NKC)
- Number of support classes
- Average number of support classes per key class (ANSC)
- Number of subsystems (NSUB)
- Class size (CS)
    - Total number of operations + number of attributes
        - Both include inherited features
- Number of operations overridden by subclass (NOO)
- Number of operations added by a subclass (NOA)
- Specialization index (SI)
    - SI = [NOO x level] / [Total class method]
- Average method size
- Average number of methods
- Average number of instance variables
- Class hierarchy nesting level

[12] Churcher, N and Shepperd, M, "Comments on 'A Metrics Suite for Object Oriented Design'", *IEEE Transactions on Software Engineering*, Vol. 21 No. 3, pp. 263-265, 1995.

[13] Lorenz, M. and Kidd, J., *Object-Oriented Software Metrics*, Prentice Hall, Englewood Cliffs, N.J., 1994.

D. PROPOSED SOFTWARE TESTING TOOL METRICS

Elaine Weyuker identified nine properties that complexity measures should possess.[14] Several of these properties can be applied to other metrics as well. These characteristics were considered during our research to develop metrics for software testing tools.

We propose that our software testing tool metric suite contain the following properties, although to varying degrees. The metrics exhibit non-coarseness in that they provide different values when applied to different testing tools. They are finite in that there are a finite number of tools for which the metrics' results in an equal value. Yet they are non-unique in that the metric may provide the same value when applied to different tools. Our metrics are designed to have an objective means of assessment rather than being based on subjective opinions of the evaluator. A testing tool metric record sheet is included in Appendix E.

1. Metrics for Tools Testing Procedural Software

These metrics are applied to the testing tool in its entirety vice a specific function performed by the tool.

a. Human Interface Design (HID)

All automated testing tools require the tester to set configurations prior to the commencement of testing. Tools with well designed human interfaces enable easy, efficient, and accurate setting of tool configuration. Factors that lead to difficult, inefficient, and inaccurate human input include multiple switching between keyboard and mouse input, requiring large amount of keyboard input overall, and individual input fields that require long strings of input. HID also accounts for easy recognition of the functionality of provided shortcut buttons.

[14].Weyuker, E., 'Evaluating Software Complexity Measures', *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1357-1365, 1988.

Equation 3.5

HID = KMS + IFPF + ALIF + (100 –BR)

Where KMS is the average number of keyboard to mouse switches per function

IFPF is the average number of input fields per function

ALIF is the average string length of input fields

BR is the percentage of buttons whose functions were identified via inspection by first time users times ten

A large HID indicates the level of difficulty to learn the tool's procedures on purchase and the likelihood of errors in using the tool over a long period of time. HID can be reduced by designing input functions to take advantage of current configurations as well as using input to recent fields as default in applicable follow on input fields. For example, if a tool requires several directories to be identified, subsequent directory path input fields could be automatically completed with previously used paths. This would require the tester to only modify the final subfolder as required vice reentering lengthy directory paths multiple times.

b. Maturity & Customer Base (MCB)

There are several providers of automated testing tools vying for the business of software testers. These providers have a wide range of experience in developing software testing tools. Tools that have achieved considerable maturity typically do so as a result of customer satisfaction in the tool's ability to adequately test their software. This satisfaction leads to referrals to other users of testing tools and an increase in the tool's customer base.

Equation 3.6

MCB = maturity + customer base + projects

Where maturity is the number of years tool (and its previous versions) have been applied in real world applications customer base is the number of customers who have more than one year of experience applying the tool

projects is the number of previous projects of similar size that used the tool

Care must be taken in evaluating maturity to ensure the tool's current version does not depart too far from the vendor's previous successful path. Customer base and projects are difficult to evaluate without relying upon information from a vendor who has a vested interest in the outcome of the measurement.

c. Tool Management (TM)

As software projects become larger and more complex, large teams are used to design, encode, and test the software. Automated testing tools should provide for several users to access the information while ensuring proper management of the information. Possible methods may include automated generation of reports to inform other testers on outcome of current tests, and different levels of access (e.g., read results, add test cases, modify/remove test cases).

Equation 3.7

TM = access levels + information control methods

Where access levels is the number of different access levels to tool information

Information control methods is the sum of the different methods of controlling tool and test information

d. Ease of Use (EU)

A testing tool must be easy to use to ensure timely, adequate, and continual integration into the software development process. Ease of use accounts for the learning time of first-time users, retainability of procedural knowledge for frequent and casual users, and operational time of frequent and casual users.

Equation 3.8

EU = LTFU + RFU + RCU + OTFU + OFCU

Where LTFU is the learning time for first users

RFU is the retainability of procedure knowledge for frequent users

RCU is the retainability of procedure knowledge for casual users

OTFU is the average operational time for frequent users

OTCU is the average operational time for casual users

e. User Control (UC)

Automated testing tools that provide users expansive control over tool operations enable testers to effectively and efficiently test those portions of the program that are considered to have a higher level of criticality, have insufficient coverage, or meet other criteria determined by the tester. UC is defined as the summation of the different portions and combinations of portions that can be tested. A tool that tests only an entire executable program would receive a low UC value. Tools that permit the tester to identify which portions of the executable will be evaluated by tester-specified test scenarios would earn a higher UC value. Tools that will be implemented by testing teams conducting a significant amount of regression testing should have a high UC value to avoid retesting of unchanged portions of code.

f. Test Case Generation (TCG)

The ability to automatically generate and readily modify test cases is desirable. Testing tools which can automatically generate test cases based on parsing the software under test are much more desirable that tools that require testers to generate their own test cases or provide significant input for tool generation of test cases. Availability of functions to create new test cases based on modification to automatically generated test cases greatly increases the tester's ability to observe program behavior under different operating conditions.

Equation 3.9

TCG = ATG + TRF

Where ATG is the level of automated test case generation as defined by:

10: fully automated generation of test cases

8: tester provides tool with parameter names & types
via user friendly methods (i.e. pull down menus)

6: tester provides tool with parameter names & types

4: tester must provide tool with parameter names, types
and range of values via user friendly methods

2: tester must provide tool with parameter names, types
and range of values

0: tester must generate test cases by hand

TRF is the level of test case reuse functionality

10: test cases may be modified by user friendly methods
(i.e. pull down menus on each test case parameter)
and saved as a new test case

8: test cases may be modified and saved as a new test case

6: test cases may be modified by user friendly methods
but cannot be saved as new test cases

4: test cases may be modified but cannot be saved as new test cases

0: test cases cannot be modified

g. Tool Support (TS)

The level of tool support is important to ensure efficient implementation of the testing tool, but it is difficult to objectively measure. Technical support should be available to testers at all times testing is being conducted, including outside traditional weekday working hours. This is especially important for the extensive amount of testing frequently conducted just prior to product release. Technical support includes help desks available telephonically or via email, and on-line users' groups monitored by vendor technical support staff. Additionally, the availability of tool documentation that is well organized, indexed, and searchable is of great benefit to users.

Equation 3.10

$$TS = ART + ARTAH + ATSD - DI$$

Where ART is the average response time during scheduled testing schedule

ARTAH is the average response time outside scheduled testing schedule

ATSD is the average time to search documentation for desired information

DI is the documentation inadequacy measured as the number of unsuccessful searches of documentation

h. Estimated Return on Investment (EROI)

A study conducted by the Quality Assurance Institute involving 1,750 test cases and 700 errors has shown that automated testing can reduce time requirements for nearly every testing stage and reduces overall testing time by approximately 75%.[15] Vendors may also be able to provide similar statistics for their customers currently using their tools.

Equation 3.11

$$EROI = (EPG \times ETT \times ACTH) + EII - ETIC + (EQC \times EHCS \times ACCS)$$

[15] *QA Quest*, The New Quality Assurance Institute, November 1995.

Where:

EPG is the Estimated Productivity Gain
ETT is the Estimated Testing Time without tool
ACTH is the Average Cost of One Testing Hour
EII is the Estimated Income Increase
ETIC is the Estimated Tool Implementation Cost
EQC is the Estimated Quality Gain
EHCS is the Estimated Hours of Customer Support per Project
ACCS is the Average Cost of One Hour of Customer Support

i. Reliability (Rel)

Tool reliability is defined as the average mean time between failures.

j. Maximum Number of Classes (MNC)

Maximum number of classes that may be included in a tool's testing project.

k. Maximum Number of Parameters (MNP)

Maximum number of parameters that may be included in a tool's testing project.

l. Response Time (RT)

Time required to conduct test case on specified size of software. Difficult to measure due to varying complexity of different programs of same size.

m. Features Support (FS)

Count of the following features:

- Extendable: tester can write functions that expand provided functions
- Database available: open database for use by testers
- Integrates with software development tools
- Provides summary reports of findings

2. Metrics for Tools Used to Test Object-Oriented Software

Studies are continuously being conducted to ascertain the validity and usefulness of other software quality metrics. A seminal study, conducted at the University of Maryland, determined that the majority of the metrics proposed by Chidamber and

Kemmerer were useful in predicting the program under test's proneness to containing faults.[16] As such, automated testing tools implemented on object-oriented software should support their metric suite with the exception of LCOM. Testing tool support of the other object-oriented software quality metrics discussed previously should also be measured. This will enable the software development manager to measure the level of support for measuring the quality of object-oriented software.

3. Difference between Procedural Testing Tool Metrics and Object-oriented Testing Tool Metrics

Through our studies, we have determined that the differences between procedural and object-oriented testing tool metrics are minimal. Metrics for testing tools aimed at object-oriented software should support the general testing tool metrics. Additionally, they should include a measurement for level of support of the object-oriented software quality metrics and one for the maximum number of classes supported by the tool.

E. PERFORM TESTS

1. LDRA Testbed

a. Set-up

LDRA Testbed was installed on a computer using Microsoft Windows 98. Projects tested were written, compiled, and executed in Microsoft Visual Studio 6.0. LDRA Testbed does not embed itself into the Visual Studio application, but does provide an icon on the desktop for easy launching of the testing tool.

b. Problems During Execution

The tool performed well once a few configuration difficulties were corrected. The installation wizard did not automatically update settings for the location of the vcvars32.bat file. In response to queries, LDRA's technical support was timely, friendly, and knowledgeable.

[16] Basili, V., etal, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *Technical Report 95-40*, University of Maryland, College Park, MD, April 1995.

2. Parasoft

a. Set-up

The following Parasoft products were installed on a computer using Microsoft Windows 2000; C++ Test, Code Wizard, and Insure++. Projects tested were written, compiled, and executed in Microsoft Visual Studio 6.0. All three products allow themselves to be integrated into the Visual Studio application. Testing operations can be conducted from either buttons added to Visual Studio toolbars or via the Tools menu on the Visual Studio menu bar.

Configuring CodeWizard: In order to use CodeWizard, you must have CodeWizard (with a valid CodeWizard license) installed on your machine.

To configure C++ Test to automatically run your classes and methods through CodeWizard, enable the Use CodeWizard option by choosing Options> Project Settings, then selecting the Use CodeWizard option in the Build Options tab.

b. Problems During Execution

Parasoft C++ Test was initially installed on a computer using Microsoft Windows 98, as had been done during earlier testing. During test execution, C++ Test consistently produced time-out errors. After speaking with technical support to identify the source of the difficulties, it was discovered that version 1.3 (June 2001) of C++ Test did not support Windows 98. After obtaining version 1.3 (July 2001) of C++ Test, it and Code Wizard and Insure++ were installed on a computer using Windows 2000. As Parasoft technical support was discussing the many features available in their products, it was determined that there was a newer version [beta version 1.3 (August 2, 2001)] available. This new version incorporates the code analysis features of Code Wizard into C++ Test.

3. Telelogic

a. Set-up

The Telelogic Tau Logiscope 5.0 testing tool suite was installed on a computer using Microsoft Windows 2000. Projects tested were written, compiled, and executed in Microsoft Visual Studio 6.0. Telelogic provides access to its functions by placing selection into the Tools menu on the Visual Studio menu bar, but does not automatically introduce graphical shortcut buttons on the Visual Studio toolbar.

b. Problems During Execution

While the example in the installation manual worked well, it did not address all the functions that are not performed by the wizard (e.g. creation of batch files). Several of the problems that we encountered could be eliminated by better organization of installation manuals, such as placing the Microsoft Visual Studio integration content at the beginning of the manual. Once integrated into Visual Studio, the tools were quite easy to use.

THIS PAGE INTENTIONALLY LEFT BLANK

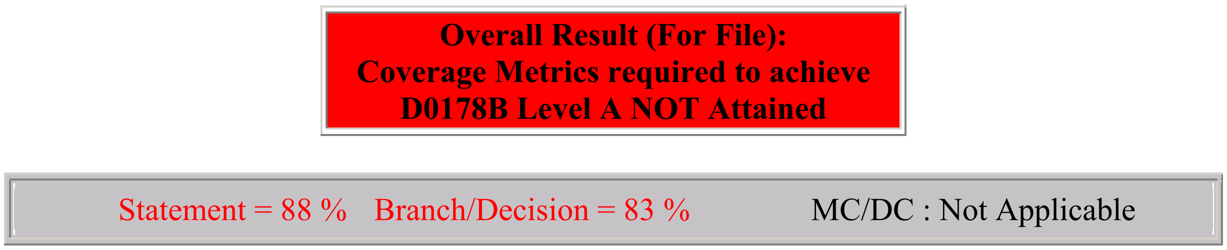
IV. ANALYSIS

A. TOOL PERFORMANCE

1. LDRA Testbed

a. Procedural

Coverage Report – In order to achieve DO178B Level A, the program must achieve 100% coverage in both statement coverage and branch coverage. The procedural program achieved an overall grade of fail because it only achieved 88% statement coverage and 83% branch coverage. 554 of a possible 629 statements were covered during the testing process. 146 out of 176 branches were covered by the testing tool. What is important to note about 88% coverage is that we only used default test settings and did not conduct additional test runs to improve our coverage. As mentioned before in the tool summary, to increase the coverage, the user must construct further sets of test data to be run with the instrumented source code. The report lists each individual line that is not executed by any testing data. The graphics below were captured directly from Testbed's report.



Overall Result (For File):
Coverage Metrics required to achieve
D0178B Level A NOT Attained

Statement = 88 % Branch/Decision = 83 % MC/DC : Not Applicable

Figure 5: Procedural Dynamic Coverage Analysis Overall Report

Executed by Runs…					Coverage(%)		
Procedure	**Executable Statements**	**Previous**	**Current**	**Combined**	**Previous**	**Current**	**Combined**
Main	629	0	554	554	0	88	88
Whole program	**629**	**0**	**554**	**554**	**0**	**88**	**88**

Figure 6: Procedural Statement Execution History Summary

Executed by Runs…					Coverage(%)		
Procedure	**Branch/ Decisions**	**Previous**	**Current**	**Combined**	**Previous**	**Current**	**Combined**
Main	176	0	146	146	0	83	83
Whole Program	**176**	**0**	**146**	**146**	**0**	**83**	**83**

Figure 7: Procedural Branch/Decision Execution History Summary

Metrics Report – Our procedural program returned a value of 130 knots and a cyclomatic complexity of sixty-one. The 130 knots signals that the procedural code is disjointed and would require somebody trying to read the code to jump back and forth between functions in order to understand what the code is attempting to accomplish. The cyclomatic complexity of sixty-one demonstrates that the program can be re-ordered to improve readability and reduce complexity. The figures below list the findings from the metrics report for Halstead Metrics, LCSAJ and Unreachability.

File	**Total Operators**	**Total Operands**	**Unique Operators**	**Unique Operands**	**Vocabulary**	**Length**	**Volume**
Seqmain.cpp	471	593	24	60	84	1064	6801

Figure 8: Halstead Metrics for procedural code

File	**Total LCSAJs**	**Reachable LCSAJs**	**Unreachable LCSAJs**	**Max. LCSAJ Density**	**Unreachable Lines**	**Unreachable Branches**
Seqmain.cpp	228	218	10	25	0	0

Figure 9: LCSAJ and Unreachability for procedural code

Quality Report – The Quality Report gives an instant view on the quality of the source code analyzed. Overall LDRA's Testbed gave the procedural program a grade of fail. It reported 109 occurrences of eighteen different violations classified as "Mandatory (Required) Standards," eleven occurrences of three different violations classified as "Checking (Mandatory/Required) Standards," and eighty occurrences of six different violations against standards considered "Optional (Advisory)." Figures 10 through Figure 13 list the different standards that were violated. If a Motor Industry Software Reliability Association (MISRA) code is violated, it is so annotated by the LDRA report.

Number of Violations	(M) Mandatory (Required) Standards	MISRA Code
1	More than 500 executable reformatted lines in file	
1	Procedure exceeds 50 reformatted lines.	
5	No brackets to loop body (added by Testbed).	MISRA 59
16	No brackets to then/else (added by Testbed).	MISRA 59
2	Use of break statement in loop.	
2	Use of continue statement.	
1	Cyclomatic complexity greater than 10.	
1	Variables declared but not used in code analyzed.	
1	UR data flow anomalies found.	MISRA 30
1	No default case in switch statement.	
21	Equality comparison of floating point.	MISRA 50
2	Boolean comparison with 0 preferred.	
1	Main must be int (void) or int (int,char*[]).	
4	Use of abort, exit, etc.	MISRA 126
36	Floating point not permitted.	
1	LCSAJ density exceeds 20.	
12	Use of a comment in a macro definition.	
1	Less than 0.10 comments per line of code	

Figure 10: Procedural Mandatory Standards Violations

Number of Violations	Annotation Code	(C)Checking (Mandatory/Required) Standards	MISRA Code
1	26 S	Infinite loop used.	
9	29 S	Use of += or -= operators found.	
1	7 C	Procedure has more than one exit point.	

Figure 11: Procedural Checking Standards Violations

Number of Violations	**(O)Optional(Advisory) Standards**	**MISRA Code**
1	Procedure contains essential knots.	
1	Procedure is not structured.	
1	DU data flow anomalies found.	
1	DD data flow anomalies found.	
17	Loop index is not declared locally.	
59	Scope of variable could be reduced	MISRA 22

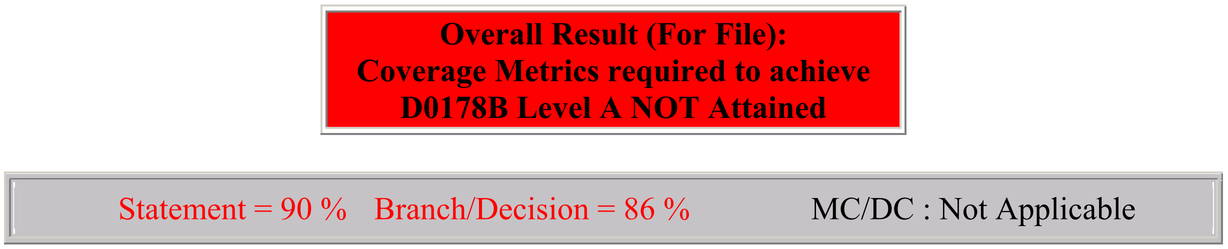
Figure 12: Procedural Optional Standards Violations

Number of procedures:	1
Number of locally uncalled procedures:	0
Maximum loop depth:	5
Total Cyclomatic Complexity:	61
Number of reformatted executable lines:	642
Number of lines of comments:	29

Figure 13: Procedural Global Basic Information

b. Functional

Coverage Report – The functional program achieved an overall grade of fail because it only achieved 90% statement coverage and 86% branch coverage. 557 of a possible 619 statements were covered during the testing process. 169 out of 196 branches were covered by the testing tool. Again, in achieving 88% coverage, we only used default test settings and did not conduct additional test runs to improve our coverage. The graphics below were captured directly from Testbed's report.



Overall Result (For File):
Coverage Metrics required to achieve
D0178B Level A NOT Attained

Statement = 90 %   Branch/Decision = 86 %   MC/DC : Not Applicable

Figure 14: Functional Dynamic Coverage Analysis Overall Report

Executed by Runs…					**Coverage(%)**		
Procedure	**Executable Statements**	**Previous**	**Current**	**Combined**	**Previous**	**Current**	**Combined**
Main	189	0	184	184	0	97	97
Pick_event	34	0	34	34	0	100	100
Arrival_event	76	0	63	63	0	83	83
Find_backoff_time	13	0	13	13	0	100	100
Resched_attempt	33	0	17	17	0	52	52
Attempt_event	105	0	80	80	0	76	76
Transmit_event	76	0	76	76	0	100	100
Departure_event	44	0	41	41	0	93	93
Calculate_results	37	0	37	37	0	100	100
Show_results	12	0	12	12	0	100	100
Whole program	**619**	**0**	**557**	**557**	**0**	**90**	**90**

Figure 15: Functional Statement Execution History Summary

Executed by Runs…					**Coverage (%)**		
Procedure	**Branch/ Decisions**	**Previous**	**Current**	**Combined**	**Previous**	**Current**	**Combined**
Main	72	0	67	67	0	93	93
Pick_event	11	0	11	11	0	100	100
Arrival_ event	27	0	16	16	0	59	59
Find_backoff _time	5	0	5	5	0	100	100
Resched_atte mpt	10	0	6	6	0	60	60
Attempt_even t	31	0	25	25	0	81	81
Transmit_eve nt	24	0	24	24	0	100	100
Departure_ev ent	10	0	9	9	9	90	90
Calculate_res ults	5	0	5	5	0	100	100
Show_results	1	0	1	1	0	100	100
Whole Program	**196**	**0**	**169**	**169**	**0**	**86**	**86**

Figure 16: Functional Branch/Decision Execution History Summary

Metrics Report – Our functional program returned a value of 109 knots and a cyclomatic complexity of fifty-five. The 109 knots signals that the functional code is disjoint and would require somebody trying to read the code to jump back and forth between functions in order to understand what the code does. The cyclomatic complexity of fifty-five demonstrates that the program can be re-ordered to improve readability and reduce complexity. The figures below list the findings from the metrics report for Halstead Metrics, LCSAJ and Unreachability.

File	**Total Operators**	**Total Operands**	**Unique Operators**	**Unique Operands**	**Vocabulary**	**Length**	**Volume**
Csma.cpp	432	546	24	55	79	978	6165

Figure 17: Halstead Metrics for functional code

File	**Total LCSAJs**	**Reachable LCSAJs**	**Unreachable LCSAJs**	**Max. LCSAJ Density**	**Unreachable Lines**	**Unreachable Branches**
Csma.cpp	237	227	10	10	0	0

Figure 18: LCSAJ and Unreachability for functional code

Quality Report – The Quality Report provides a view of the quality of the source code. Overall LDRA's Testbed gave the functional program a grade of fail. It reported 115 occurrences of eighteen different violations classified as "Mandatory (Required) Standards," fourteen occurrences of four different violations classified as "Checking (Mandatory/Required) Standards," and thirty-six occurrences of six different violations against standards considered "Optional (Advisory)."

Number of Violations	(M) Mandatory (Required) Standards	MISRA Code
1	More than 500 executable reformatted lines in file	
5	Procedure exceeds 50 reformatted lines.	
4	No brackets to loop body (added by Testbed).	MISRA 59
11	No brackets to then/else (added by Testbed).	MISRA 59
2	Use of break statement in loop.	
2	Use of continue statement.	
2	Cyclomatic complexity greater than 10.	
1	UR data flow anomalies found.	MISRA 30
1	No default case in switch statement.	
18	Equality comparison of floating point.	MISRA 50
9	Empty parameter list to procedure/function	
9	Procedure definition has no associated prototype	
2	Boolean comparison with 0 preferred.	
1	Main must be int (void) or int (int,char*[ ]).	
4	Use of abort, exit, etc.	MISRA 126
30	Floating point not permitted.	
12	Use of a comment in a macro definition.	
1	Less than 5 comments in procedure header	

Figure 19: Functional Mandatory Standards Violations

Number of Violations	Annotation Code	(C)Checking (Mandatory/Required) Standards	MISRA Code
1	26 S	Infinite loop used.	
9	29 S	Use of += or -= operators found.	
2	7 C	Procedure has more than one exit point.	
2	18D	Identifier name reused	MISRA 12

Figure 20: Functional Checking Standards Violations

Number of Violations	**(O)Optional(Advisory) Standards**	**MISRA Code**
2	Procedure contains essential knots.	
2	Procedure is not structured.	
1	DU data flow anomalies found.	
4	DD data flow anomalies found.	
9	Globals used inside procedure	
18	Scope of variable could be reduced	MISRA 22

Figure 21: Functional Optional Standards Violations

Number of procedures:	10
Number of locally uncalled procedures:	0
Maximum loop depth:	4
Total Cyclomatic Complexity:	55
Number of reformatted executable lines:	629
Number of lines of comments:	277

Figure 22: Functional Global Basic Information

c. Object-Oriented

Coverage Report – Technical difficulties prevented of coverage data for the object-oriented program.

Metrics Report – The object-oriented program returned a value of fifty-six knots and a cyclomatic complexity of forty-seven. The fifty-six knots indicates that the object-oriented code is disjoint and would require somebody trying to read the code to jump back and forth between functions in order to understand what the code is attempting to accomplish. The cyclomatic complexity of forty-seven indicates that the program can be re-ordered to improve readability and reduce complexity. The figures below list the findings from the metrics report for Halstead Metrics, LCSAJ and Unreachability.

File	Total Operators	Total Operands	Unique Operators	Unique Operands	Vocabulary	Length	Volume
Simulation.cpp	363	734	25	122	147	1097	7898

Figure 23: Halstead Metrics for object-oriented code

File	Total LCSAJs	Reachable LCSAJs	Unreachable LCSAJs	Max. LCSAJ Density	Unreachable Lines	Unreachable Branches
Simulation.cpp	188	183	5	25	0	0

Figure 24: Object-Oriented LCSAJ and Unreachability for procedural code

Quality Report – The Quality Report gives an instant view on the quality of the source code analyzed. Overall LDRA's Testbed gave the object-oriented program a grade of fail. It reported 401 occurrences of thirty-one different violations classified as "Mandatory (Required) Standards," 102 occurrences of nine different violations classified as "Checking (Mandatory/Required) Standards," and seventy-five occurrences of nine different violations against standards considered "Optional (Advisory)."

d. Reporting Characteristics

LDRA's Testbed has numerous report formats to support many different decision processes. The static call-graph displays the connections between methods with each method shown in a color that signifies the status of that method's testing.

Number of Violations	(M) Mandatory (Required) Standards	MISRA Code
1	More than 500 executable reformatted lines in file	
8	Procedure exceeds 50 reformatted lines.	
4	No brackets to loop body (added by Testbed).	MISRA 59
22	No brackets to then/else (added by Testbed).	MISRA 59
2	Use of break statement in loop.	
2	Use of continue statement.	
2	Cyclomatic complexity greater than 10.	
1	Function does not return a value on all paths.	
1	Variables declared but not used in code analyzed.	
1	UR data flow anomalies found.	MISRA 30
3	Parameters do not match expected actions.	
3	Attempt to change parameter passed by value.	
1	Function has no return statement.	
1	No default case in switch statement.	
7	Equality comparison of floating point.	MISRA 50
21	Empty parameter list to procedure/function.	
2	Macro contains unacceptable items	MISRA 90 DERA 157
9	Procedure definition has no associated prototype	MISRA 71
6	Expression is not Boolean.	MISRA 35,36,49
2	Boolean comparison with 0 preferred.	
1	Main must be int (void) or int (int,char*[]).	
6	Use of abort, exit, etc.	MISRA 126
1	Array has no bounds specified.	
23	Parameter has same name as global variable.	MISRA 21 DERA 128
4	Name reused in inner scope.	MISRA 21 DERA 128
4	Prototype and Definition name mismatch.	MISRA 74
162	Floating point not permitted.	
2	No return type for function/procedure	MISRA 75
1	LCSAJ density exceeds 20.	
24	Use of a comment in a macro definition.	
4	Nested header files found.	
40	Less than 5 comments in procedure header	
30	Less than 0.10 comments per line of code	

Figure 25: Object-Oriented Mandatory Standards Violations

Number of Violations	**Annotation Code**	**(C) Checking (Mandatory/Required) Standards**	**MISRA Code**
1	23S	Procedure is not called in text analyzed.	
1	26 S	Infinite loop used.	
9	29 S	Use of += or -= operators found.	
7	1D	Unused procedure parameter.	
11	9D	Defined parameter has possible clear path.	
9	7 C	Procedure has more than one exit point.	
20	18D	Identifier name reused	MISRA 12
33	19D	Procedure called before being defined.	MISRA 20,71
11	23D	Function has parameter side effects	MISRA 33,46

Figure 26: Object-Oriented Checking Standards Violations

Number of Violations	**(O)Optional(Advisory) Standards**	**MISRA Code**
1	Procedure contains essential knots.	
1	Procedure is not structured.	
6	DU data flow anomalies found.	
11	DD data flow anomalies found.	
13	Globals used inside procedure.	
7	Referenced parameter has possible clear path.	
2	Loop index is not declared locally.	
15	Name declared in another namespace (MR).	MISRA 12
19	Scope of variable could be reduced	MISRA 22

Figure 27: Object-Oriented Optional Standards Violations

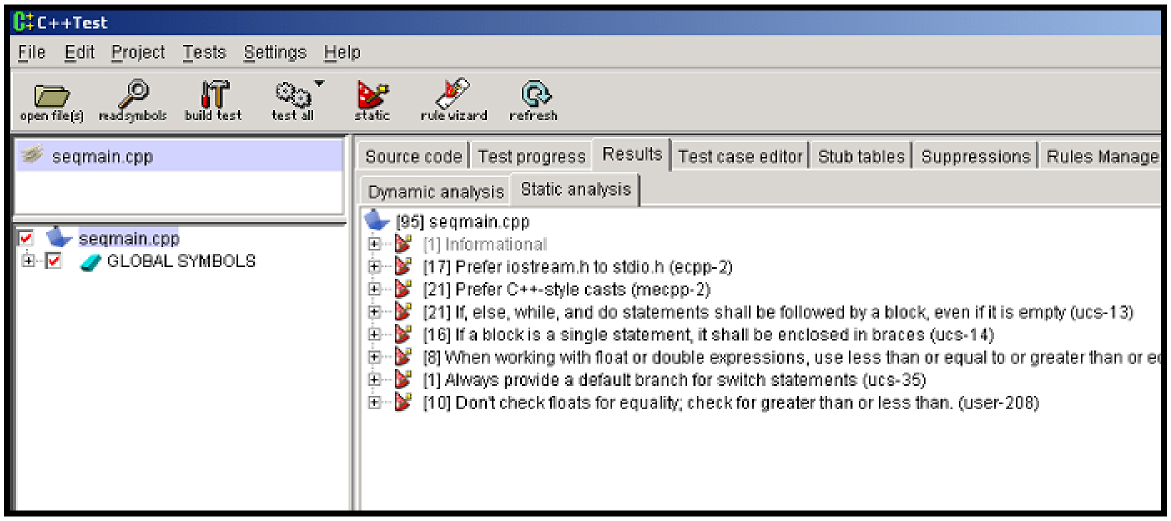
Number of procedures:	49
Number of locally uncalled procedures:	1
Maximum loop depth:	3
Total Cyclomatic Complexity:	83
Number of reformatted executable lines:	968
Number of lines of comments:	1017

Figure 28: Object-Oriented Global Basic Information

2. Parasoft

a. *Procedural*

Parasoft C++ (with integrated Code Wizard) detected 95 occurrences of eight different rule violations (Figure 29).


C++Test
File Edit Project Tests Settings Help
open file(s) readsymbols build test test all static rule wizard refresh
seqmain.cpp
seqmain.cpp
GLOBAL SYMBOLS
Source code Test progress Results Test case editor Stub tables Suppressions Rules Manage
Dynamic analysis Static analysis

[95] seqmain.cpp
[1] Informational
[17] Prefer iostream.h to stdio.h (ecpp-2)
[21] Prefer C++-style casts (mecpp-2)
[21] If, else, while, and do statements shall be followed by a block, even if it is empty (ucs-13)
[16] If a block is a single statement, it shall be enclosed in braces (ucs-14)
[8] When working with float or double expressions, use less than or equal to or greater than or e
[1] Always provide a default branch for switch statements (ucs-35)
[10] Don't check floats for equality; check for greater than or less than. (user-208)

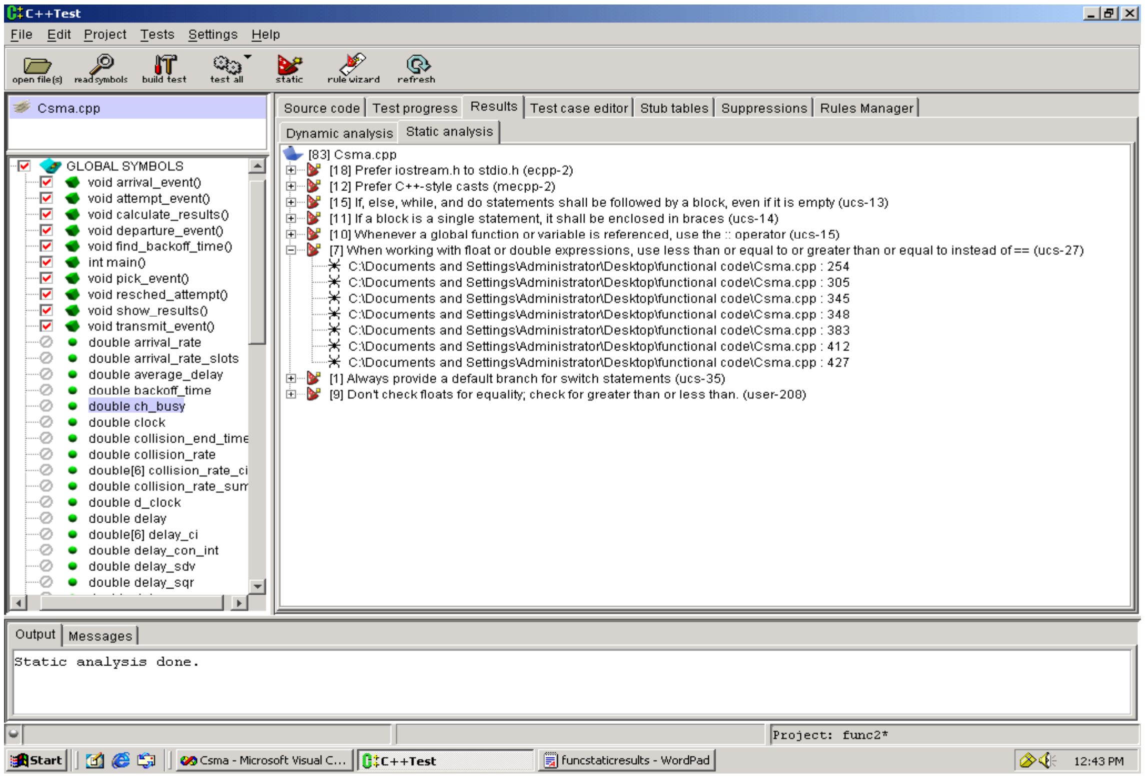
Figure 29: Parasoft Procedural Static Analysis Report

b. *Functional*

Parasoft C++ (with integrated Code Wizard) detected eighty-three occurrences of eight different rule violations (Figure 30) during static analysis of the functional version of the source code. Of the 328 test cases conducted, 321 passed and seven reported time-out errors (Figure 31).

c. *Object-Oriented*

Parasoft C++ (with integrated Code Wizard) detected 122 occurrences of 12 different rule violations during static analysis of the object-oriented version of the source code. The findings for the six classes are summarized in Figure 32. Of the seventy-one test cases conducted, fifty passed and twenty-one reported access violation exception errors (Figure 33). Insure++ reported thirty-nine outstanding memory references.


C++Test
File Edit Project Tests Settings Help
open file(s) read symbols build test test all static rule wizard refresh
Source code | Test progress | Results | Test case editor | Stub tables | Suppressions | Rules Manager
Dynamic analysis | Static analysis
Output | Messages
Start
Csma - Microsoft Visual C...
C++Test
funcstaticresults - WordPad
12:43 PM

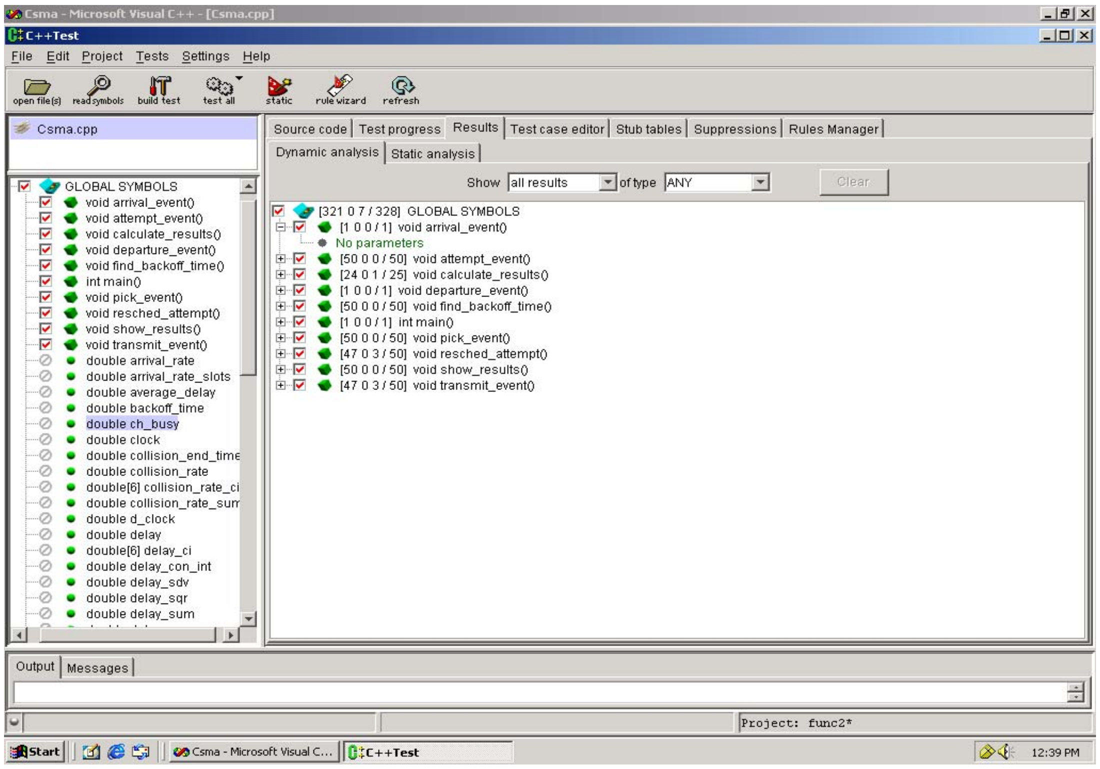
```
Csma.cpp

GLOBAL SYMBOLS
  void arrival_event()
  void attempt_event()
  void calculate_results()
  void departure_event()
  void find_backoff_time()
  int main()
  void pick_event()
  void resched_attempt()
  void show_results()
  void transmit_event()
  double arrival_rate
  double arrival_rate_slots
  double average_delay
  double backoff_time
  double ch_busy
  double clock
  double collision_end_time
  double collision_rate
  double[6] collision_rate_ci
  double collision_rate_sum
  double d_clock
  double delay
  double[6] delay_ci
  double delay_con_int
  double delay_sdv
  double delay_sqr
```

```
[83] Csma.cpp
  [18] Prefer iostream.h to stdio.h (ecpp-2)
  [12] Prefer C++-style casts (mecpp-2)
  [15] If, else, while, and do statements shall be followed by a block, even if it is empty (ucs-13)
  [11] If a block is a single statement, it shall be enclosed in braces (ucs-14)
  [10] Whenever a global function or variable is referenced, use the :: operator (ucs-15)
  [7] When working with float or double expressions, use less than or equal to or greater than or equal to instead of == (ucs-27)
      C:\Documents and Settings\Administrator\Desktop\functional code\Csma.cpp : 254
      C:\Documents and Settings\Administrator\Desktop\functional code\Csma.cpp : 305
      C:\Documents and Settings\Administrator\Desktop\functional code\Csma.cpp : 345
      C:\Documents and Settings\Administrator\Desktop\functional code\Csma.cpp : 348
      C:\Documents and Settings\Administrator\Desktop\functional code\Csma.cpp : 383
      C:\Documents and Settings\Administrator\Desktop\functional code\Csma.cpp : 412
      C:\Documents and Settings\Administrator\Desktop\functional code\Csma.cpp : 427
  [1] Always provide a default branch for switch statements (ucs-35)
  [9] Don't check floats for equality; check for greater than or less than. (user-208)
```

```
Static analysis done.
```

```
Project: func2*
```

Figure 30: Parasoft Functional Static Analysis Report


Csma - Microsoft Visual C++ - [Csma.cpp]
C++Test
File Edit Project Tests Settings Help
open file(s) read symbols build test test all static rule wizard refresh
Csma.cpp
Source code | Test progress | Results | Test case editor | Stub tables | Suppressions | Rules Manager
Dynamic analysis | Static analysis
Show all results of type ANY Clear
Output | Messages
Project: func2*
Start
Csma - Microsoft Visual C...
C++Test
12:39 PM

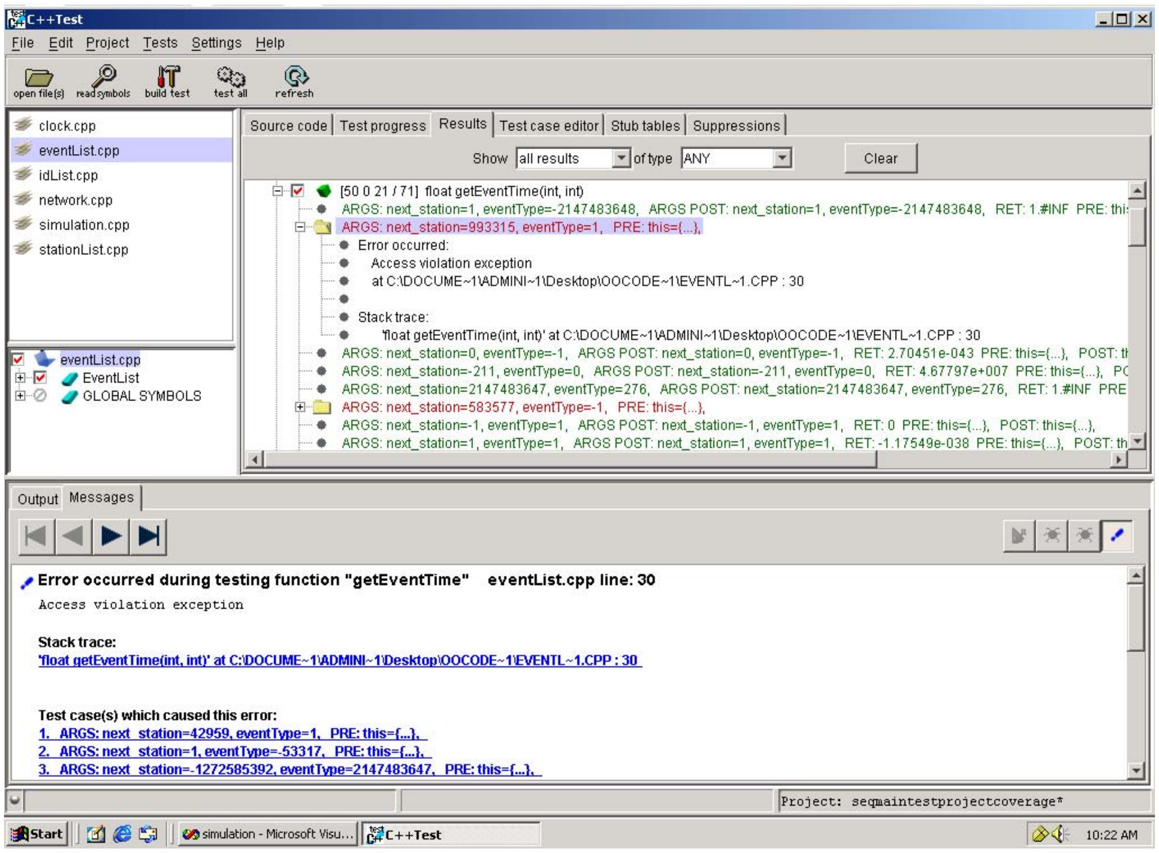
```
GLOBAL SYMBOLS
  void arrival_event()
  void attempt_event()
  void calculate_results()
  void departure_event()
  void find_backoff_time()
  int main()
  void pick_event()
  void resched_attempt()
  void show_results()
  void transmit_event()
  double arrival_rate
  double arrival_rate_slots
  double average_delay
  double backoff_time
  double ch_busy
  double clock
  double collision_end_time
  double collision_rate
  double[6] collision_rate_ci
  double collision_rate_sum
  double d_clock
  double delay
  double[6] delay_ci
  double delay_con_int
  double delay_sdv
  double delay_sqr
  double delay_sum
```

```
[321 0 7 / 328]  GLOBAL SYMBOLS
  [1 0 0 / 1]  void arrival_event()
     No parameters
  [50 0 0 / 50]  void attempt_event()
  [24 0 1 / 25]  void calculate_results()
  [1 0 0 / 1]  void departure_event()
  [50 0 0 / 50]  void find_backoff_time()
  [1 0 0 / 1]  int main()
  [50 0 0 / 50]  void pick_event()
  [47 0 3 / 50]  void resched_attempt()
  [50 0 0 / 50]  void show_results()
  [47 0 3 / 50]  void transmit_event()
```

Figure 31: Parasoft Functional Dynamic Analysis Report

	Clock	**EventList**	**IdList**	**Network**	**Simulation**	**StationList**	**Total**
Ecpp-2	3		1		11	1	16
Ecpp-12		2		1			3
Mecpp-2		2			18		20
Ucs-13		2	1	3	17	1	24
Ucs-14		2	1	3	14		20
Ucs-15	3	5	1		9	1	19
Ucs-23			2			3	5
Ucs-27				2	3		5
Ucs-32					1		1
Ucs-35					1		1
Ucs-37				2			2
User-208				2	4		6

Figure 32: Parasoft Object-Oriented Static Analysis Summary Table



```
[50 0 21 / 71]  float getEventTime(int, int)
    ARGS: next_station=1, eventType=-2147483648,  ARGS POST: next_station=1, eventType=-2147483648,  RET: 1.#INF  PRE: thi
    ARGS: next_station=993315, eventType=1,  PRE: this={...},
        Error occurred:
            Access violation exception
            at C:\DOCUME~1\ADMINI~1\Desktop\OOCODE~1\EVENTL~1.CPP : 30

        Stack trace:
            'float getEventTime(int, int)' at C:\DOCUME~1\ADMINI~1\Desktop\OOCODE~1\EVENTL~1.CPP : 30
    ARGS: next_station=0, eventType=-1,  ARGS POST: next_station=0, eventType=-1,  RET: 2.70451e-043  PRE: this={...},  POST: th
    ARGS: next_station=-211, eventType=0,  ARGS POST: next_station=-211, eventType=0,  RET: 4.67797e+007  PRE: this={...},  PO
    ARGS: next_station=2147483647, eventType=276,  ARGS POST: next_station=2147483647, eventType=276,  RET: 1.#INF  PRE
    ARGS: next_station=583577, eventType=-1,  PRE: this={...},
    ARGS: next_station=-1, eventType=1,  ARGS POST: next_station=-1, eventType=1,  RET: 0  PRE: this={...},  POST: this={...},
    ARGS: next_station=1, eventType=1,  ARGS POST: next_station=1, eventType=1,  RET: -1.17549e-038  PRE: this={...},  POST: th
```

```
Error occurred during testing function "getEventTime"    eventList.cpp line: 30
Access violation exception

Stack trace:
'float getEventTime(int, int)' at C:\DOCUME~1\ADMINI~1\Desktop\OOCODE~1\EVENTL~1.CPP : 30


Test case(s) which caused this error:
1.  ARGS: next_station=42959, eventType=1,  PRE: this={...},
2.  ARGS: next_station=1, eventType=-53317,  PRE: this={...},
3.  ARGS: next_station=-1272585392, eventType=2147483647,  PRE: this={...},
```

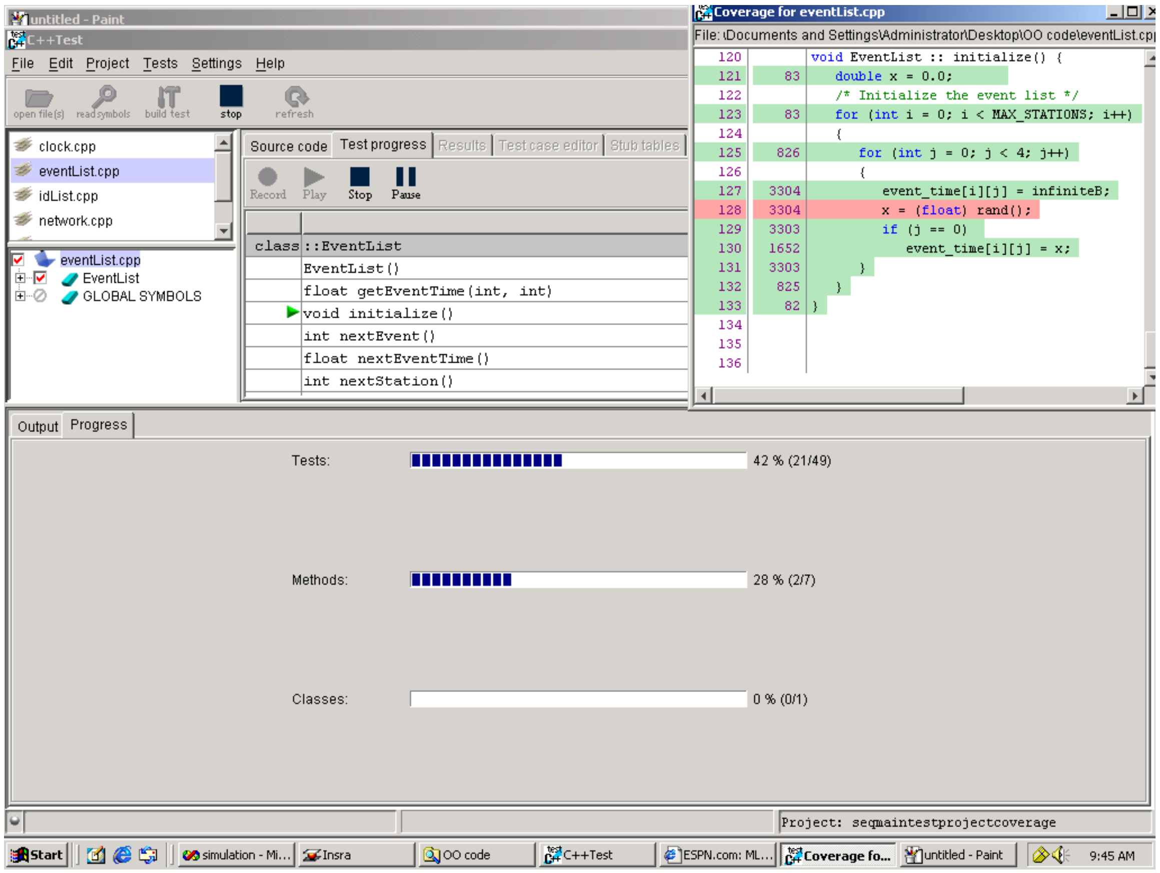
```
Project: seqmaintestprojectcoverage*
```

Figure 33: Parasoft Object-Oriented Dynamic Report

d. Reporting Characteristics

C++Test, CodeWizard, and Insure++ provide itemized reports of discovered errors, but do not provide extensive summary reports. Thus, the reports generated by these tools are quite different than those provided by LDRA.

During the execution of testing C++Test reports the progress using bar graphs to indicate the number and percentage of methods and tests conducted. Additionally, if coverage is enabled the tools will highlight the lines of code which have been tested. Figure 34 demonstrates the reports shown during test execution.



```
class::EventList
EventList()
float getEventTime(int, int)
void initialize()
int nextEvent()
float nextEventTime()
int nextStation()
```

Coverage for eventList.cpp

File: \Documents and Settings\Administrator\Desktop\OO code\eventList.cpp

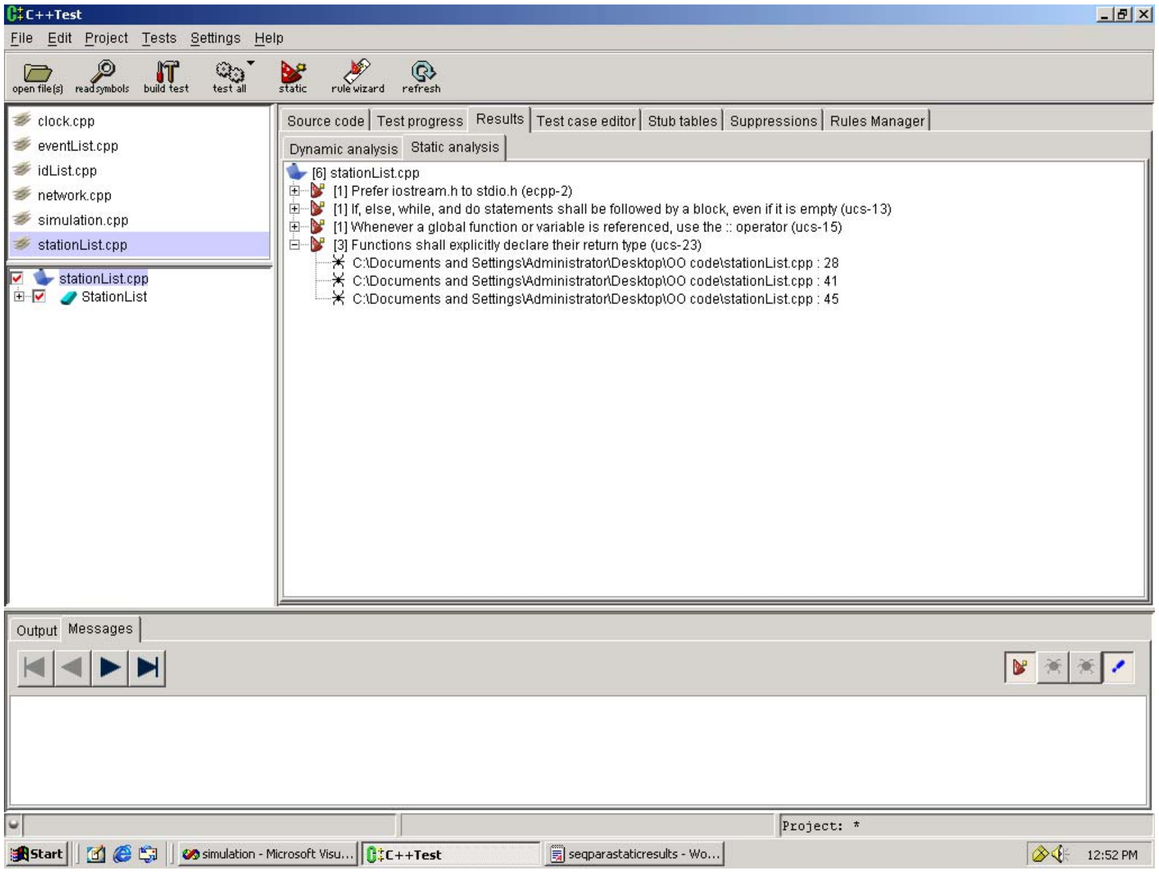
```
        void EventList :: initialize() {
  83       double x = 0.0;
           /* Initialize the event list */
  83       for (int i = 0; i < MAX_STATIONS; i++)
           {
 826          for (int j = 0; j < 4; j++)
              {
3304             event_time[i][j] = infiniteB;
3304             x = (float) rand();
3303             if (j == 0)
1652                event_time[i][j] = x;
3303          }
 825       }
  82   }
```

```
Tests:     42 % (21/49)
Methods:   28 % (2/7)
Classes:   0 % (0/1)
```

Project: seqmaintestprojectcoverage

Figure 34: Parasoft Testing Status Report

Results of the static analysis conducted upon the source code are reported under the "Static analysis" tab under the "Results" tab as shown in Figure 35. The number in square braces next to the file name indicates the total number of occurrences of coding rule violations within that file. The next line indicates the number of occurrences of violations of a specific coding rule. Expanding the line reveals the location (i.e., source code line number) of each occurrence of the violation.


C++Test
File Edit Project Tests Settings Help
open file(s) read symbols build test test all static rule wizard refresh
Source code Test progress Results Test case editor Stub tables Suppressions Rules Manager
Dynamic analysis Static analysis
Output Messages
Project: *
Start
simulation - Microsoft Visu...
C++Test
seqparastaticresults - Wo...
12:52 PM

```
clock.cpp
eventList.cpp
idList.cpp
network.cpp
simulation.cpp
stationList.cpp

stationList.cpp
  StationList
```

```
[6] stationList.cpp
  [1] Prefer iostream.h to stdio.h (ecpp-2)
  [1] If, else, while, and do statements shall be followed by a block, even if it is empty (ucs-13)
  [1] Whenever a global function or variable is referenced, use the :: operator (ucs-15)
  [3] Functions shall explicitly declare their return type (ucs-23)
      C:\Documents and Settings\Administrator\Desktop\OO code\stationList.cpp : 28
      C:\Documents and Settings\Administrator\Desktop\OO code\stationList.cpp : 41
      C:\Documents and Settings\Administrator\Desktop\OO code\stationList.cpp : 45
```

Figure 35: Parasoft Static Analysis Report

Results of the dynamic analysis conducted on the source code are reported under the "Dynamic analysis" tab under the "Results" tab as shown in Figure 31. Each line indicates the status of testing for an individual method. The numbers in the square braces on the first line indicate the following information (Figure 36):

- **OK:** The number of test cases that in which the method returned and had the correct return value and/or post-condition
- **Failed:** The number of test cases in which the test did not have the correct return value or post-condition
- **Error:** The number of test cases in which the method crashed
- **Total:** The total number of test cases used

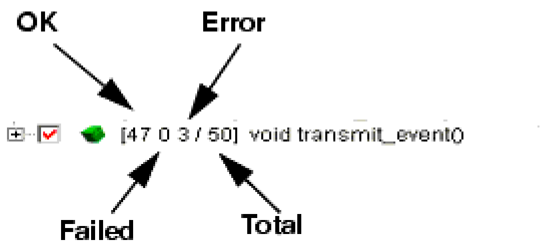
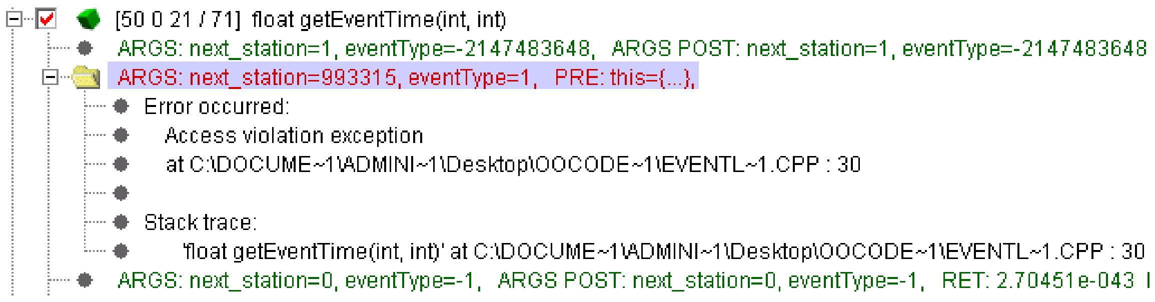


Figure 36: Parasoft Method Test Status

Clicking on a test case's results will cause its branch to expand. If a test case passes, it will display (in green) the number of times it was executed (in braces) and its arguments, returns, preconditions, and post-conditions as shown in the second line of Figure 37.



```
[50 0 21 / 71]  float getEventTime(int, int)
  ARGS: next_station=1, eventType=-2147483648,  ARGS POST: next_station=1, eventType=-2147483648
  ARGS: next_station=993315, eventType=1,  PRE: this={...},
    Error occurred:
      Access violation exception
      at C:\DOCUME~1\ADMINI~1\Desktop\OOCODE~1\EVENTL~1.CPP : 30

    Stack trace:
        'float getEventTime(int, int)' at C:\DOCUME~1\ADMINI~1\Desktop\OOCODE~1\EVENTL~1.CPP : 30
  ARGS: next_station=0, eventType=-1,  ARGS POST: next_station=0, eventType=-1,  RET: 2.70451e-043  I
```

Figure 37: Parasoft Method Test Case Detailed Report

If a test case had an error or failed, expanding its branch will display the number of times it was executed (in braces, in red), its arguments, returns, preconditions, post-conditions, and details about the type of exception or error found. It also indicates the line number at which the exception or error occurred as shown in the lines three through nine in Figure 37.

3. Logiscope

a. Procedural

Telelogic's Logiscope reported 218 occurrences of fourteen different programming rule violations as listed in Figure 38. If a rule is violated, it is so annotated

Rule Mnemonic	Rule Name	State	Lines
ansi	Function declaration in ANSI syntax	Violated	55,
asscal	Assignment in function calls	Not Violated	-
brkcont	Break and continue forbidden	Violated	168, 170, 200, 231,
cast	Prefer C++-style Casts	Violated	67, 121, 199, 221, 225, 225, 243, 264, 266, 299, 301, 307, 309, 327, 329, 383,
const	Literal constants	Violated	25, 25, 25, 25, 25, 25, 28, 57, 58, 59, 60, 61, 63, 64, 64, 64, 67, 68, 68, 68, 71, 73, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 95, 97, 107, 118, 134, 146, 148, 177, 183, 192, 200, 234, 236, 238, 238, 247, 247, 252, 254, 255, 260, 262, 267, 267, 271, 275, 277, 278, 283, 287, 291, 296, 298, 302, 302, 310, 310, 317, 319, 320, 321, 325, 330, 330, 334, 338, 340, 341, 348, 351, 362, 364, 367, 369, 374, 389, 390, 391, 392, 393, 394, 398, 400, 406, 411, 418, 419, 420, 421, 422, 423, 424, 425, 426,
ctrlblock	Blocks in control statements	Violated	104, 113, 123, 151, 170, 192, 200, 231, 254, 267, 270, 277, 287, 287, 302, 310, 330, 333, 340, 357, 369,
exprcplx	Expressions complexity	Violated	64, 67, 225, 266, 301, 309, 329, 383, 408, 413,
headercom	Function and class header comments	Violated	55,
Headercom	Module header comments	Violated	1,
identl	Identifier length	Violated	34, 35, 36, 36, 52, 52, 52, 52,
sgdecl	A single variable per declaration	Violated	34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 52, 53,
slcom	Use // comments	Violated	22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 45, 46, 48, 49, 93, 103, 129, 154, 159, 197, 207, 283, 348, 356,
slstat	One statement per line	Violated	104, 113, 123, 151, 170, 200, 231, 267, 287, 302, 310, 330,
swdef	default within switch	Violated	152,

Figure 38: Logiscope Procedural Rule Violations

in red within the “State” column followed by a listing of source code line numbers where the rule violation occurs in the “Lines” column. If a rule is not violated, it is so stated in green in the “State” column. Only one rule which was not violated is shown for demonstration purposes, all other rules which were not violated were removed from the table for space concerns.

b. Functional

Technical difficulties were experienced in trying to conduct tests on the functional version of the software. Test results were inconclusive.

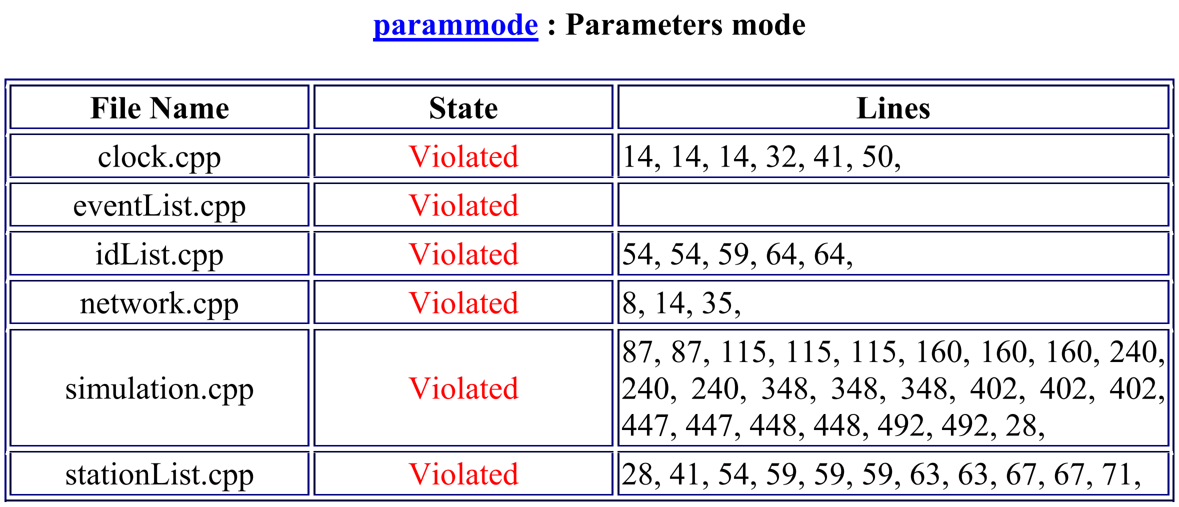
c. Object-Oriented

Logiscope identified 372 occurrences of twenty different rules violations in the object-oriented version of the network simulation program. The reports are in the same format as for procedural with each file’s violations displayed in a separate table. Technical difficulties were encountered with the Quality Report. Function level attributes were measured to be in the “Excellent” or “Good” range for more than 90% of the functions.

d. Reporting Characteristics

Logiscope provides its reports in HTML format, which allows for easy navigation within the reports. The Rules report contains the table listed in Figure 38. Additionally, it includes a separate table for each rule listing the occurrences of violations for each file (Figure 39). There is an additional “Synthesis Table” which creates a matrix summarizing the number of violations of each rule per each file. Each mention of a rule is hyperlinked to a detailed explanation of the rule at the bottom of the report. File names are linked to the table which lists the violations within that report. The reports also list the date and time the analysis was last conducted on each file. This feature assists in the management of the testing reports.

The Quality report is also in HTML format and provides similar hyperlink features as the Rules report. When analyzing object-oriented programs, Logiscope



parammode : Parameters mode

File Name	State	Lines
clock.cpp	Violated	14, 14, 14, 32, 41, 50,
eventList.cpp	Violated	
idList.cpp	Violated	54, 54, 59, 64, 64,
network.cpp	Violated	8, 14, 35,
simulation.cpp	Violated	87, 87, 115, 115, 115, 160, 160, 160, 240, 240, 240, 348, 348, 348, 402, 402, 402, 447, 447, 448, 448, 492, 492, 28,
stationList.cpp	Violated	28, 41, 54, 59, 59, 59, 63, 63, 67, 67, 71,

Figure 39: Logiscope Rule Violation Report

provides reports on three levels: application, class, and function. At the application level, the project is given a Maintainability score of Excellent, Good, Fair or Poor. The score is based on the project's scoring in four areas: Analyzability, Changeability, Stability, and Testability. All five areas are hyperlinked to the functions the tool uses to calculate the scores. The scoring tables are followed by a table listing over twenty application level metrics including Method Inheritance Factor, Method Hiding Factor, Polymorphism Factor, Coupling Factor, and many others including cyclomatic complexity measures.

The Class level section of the report displays the same attributes as the Application Level with the addition of three metrics: reusability, usability, and specializability. Again, each is hyperlinked to explanations of the methods for determining each attribute's values.

B. TESTING TOOL METRICS

During the application of the three testing-tool suites on the three software versions, measurements were taken to calculate the testing-tool metrics.

1. Human Interface Design

To calculate the human-interface design (HID) metric, measurements were taken during three operations: establishing test project, conducting test project, and viewing testing results.

While conducting the operations with the LDRA tools, there were six occasions that required the user to transfer from the keyboard to the mouse or vice versa. Dividing this number by the number of operations (three) results in an average of two keyboard-to-mouse switches (KMS). There were fifteen input fields resulting in five average input fields per functions (IFPF). Eleven of the input fields required only mouse clicks and six required entry of strings totaling eighty-three characters. The average length of input fields (ALIF) was calculated by dividing the sum of these inputs (ninety-four) by the number of input fields (sixteen) resulting in an ALIF of six. In attempting to identify the functions of sixteen buttons, eleven were identified correctly. The percentage of 68.75 was subtracted from 100, divided by ten, and rounded to the nearest integer to arrive at a button recognition factor (BR) of three. The sum of KMS, IFPF, ALIF, and BR earns LDRA a HID score of sixteen.

The same operations were performed with the Telelogic products. There were fifteen occasions that required the user to transfer from the keyboard to the mouse or vice versa. Dividing this number by the number of operations (three) results in an average of five keyboard-to-mouse switches (KMS). There were twenty-four input fields resulting in eight average input fields per functions (IFPF). Seventeen of the input fields required only mouse clicks and seven required entry of strings totaling 146 characters. The average length of input fields (ALIF) was calculated by dividing the sum of these inputs (163) by the number of input fields (twenty-four) resulting in an ALIF of seven. In attempting to identify the functions of ten buttons, four were identified correctly. The percentage of forty was subtracted from 100 and divided by ten to arrive at a button recognition factor (BR) of six. The sum of KMS, IFPF, ALIF, and BR earns LDRA a HID score of twenty-six.

Repeating the operations with the Parasoft tools, there were six occasions that required the user to transfer from the keyboard to the mouse or vice versa. Dividing this number by the number of operations (three) results in an average of two keyboard-to-mouse switches (KMS). There were twenty-two input fields resulting in eight average input fields per functions (IFPF). Sixteen of the input fields required only mouse clicks and six required entry of strings totaling sixty-nine characters. The average length of input fields (ALIF) was calculated by dividing the sum of these inputs (eighty-seven) by the number of input fields (twenty-two) resulting in an ALIF of four. In attempting to identify the functions of sixteen buttons, fourteen were identified correctly. The percentage of seventy-five was subtracted from 100, divided by ten and rounded to the nearest integer to arrive at a button recognition factor (BR) of three. The sum of KMS, IFPF, ALIF, and BR earns LDRA a HID score of seventeen. The HID scores for the three tool suites are shown in Table 1.

Table 1. Human-Interface Design Scores

	Parasoft	**Telelogic**	**LDRA**
KMS	2	5	2
IFPF	8	8	5
ALIF	4	7	6
BR	3	6	3
HID	17	26	16

2. Test Case Generation

Test case generation (TCG) measurements were also obtained for each group of tools. LDRA does not automatically generate test cases but does provide user-friendly features such as pull-down menus for created test cases therefore it was assigned an eight for its level of automated test case generation (ATG). LDRA offers user-friendly features to allow for modifying existing test cases so it earned a score of ten for its level of test case reuse functionality (TRF). Telelogic does provide automatic test case generation so it earned an ATG score of ten. However, authors were unable to find reference to test case modification within the testing tool application or documentation. Therefore, it was not assigned a TRF value. Parasoft also provides automatic test case generation and user-friendly test-case-reuse functions, resulting in scoring ten in both ATG and TRF. The sums of the ATG and TRF are given in Table 2.

Table 2. Test-Case Generation Scores

	Parasoft	**Telelogic**	**LDRA**
ATG	10	10	8
TRF	10	0	10
TCG	20	10	18

3. Reporting Features

The Reporting Features (RF) metric is determined by one point for automatically generating summary reports and one point for producing reports in a format (e.g., HTML or ASCII text documents) that are viewable outside the application. LDRA and Telelogic automatically generate summary reports formatted in HTML earning a RF measure of two for each vendor. Parasoft also automatically produces summary reports, but they must be viewed within the Parasoft testing application. Therefore, Parasoft's RF measure is one.

4. Response Time

Each tool performed well with regards to response time. LDRA averaged twenty-five minutes in performing its tests. Telelogic averaged approximately thirty-five minutes. Parasoft averaged forty-three minutes.

5. Feature Support

The Feature Support (FS) is the count of the following features that are supported: tool supports user-written functions extending tool functionality, stores information in a database open to the user, and integrates itself into software development tools. LDRA supports all these features resulting in a FS of three. Telelogic supports an open database and integration, but the authors were unable to determine its extendibility support. Telelogic earned a FS score of two. Parasoft integrates itself with software development tools, but no information regarding the two other features was available. Therefore, Parasoft's FS value was assigned a value of one.

6. Metric Suites Supported

The Metric Suites Supported (MSS) metric is based on the tool's support of three different software quality metric suites: McCabe, function points, and Halstead. Parasoft does not report on any of these metrics, and hence, it is assigned a value of zero. Telelogic and LDRA report on McCabe and Halstead, but not function points, earning each a MSS value of two. LDRA is developing the capability to report function-point metrics.

7. Maximum Number of Classes

No tool reported a limit on the number of classes it could support when testing object-oriented programs. Even so, this metric should remain within the testing tool metric. It could be severely damaging to a software development project's success if a tool were selected and implemented only to discover it could not support the number of classes contained in the project.

8. Object-Oriented Software Quality Metrics

The Object-oriented Software Quality Metrics is the count of various object-oriented software metrics including those from the metrics suites created by Chidamber & Kemmerer, Lie & Henry, Lorenz & Kidd, and Henry & Kafura. Parasoft does not report any of these metrics, resulting in no score. Telelogic supports the Chidamber & Kemmerer suite, the Le & Henry suite, as well as several from the Lorenz & Kidd suite, thus earning an OOSWM value of twelve. LDRA also supports metrics from several of the suites warranting a score of eleven. Measurement of this metric is complicated through tools referring to measurements by titles not matching those listed in the suites. Project managers should consult tool documentation or vendor representatives if a desired metric does not appear to be supported.

9. Tool Management

None of the three testing tool suites provide different access levels or other information control methods. Tool management must be controlled via computer policies implemented in the operating system and other applications outside of the suite of testing tools.

10. User Control

All tools offered extensive user control of which portions of the code would be tested by a specified test case. Each allowed the user to specify a function, class, or project, or any combination of the three, to be tested.

11. Other Testing Tool Metrics

The remaining testing tool metrics require execution of extensive experiments or input from tool vendors. The scope of our research prevents conducting detailed experiments. Along with insufficient input from the vendors, this prevents analysis of the remaining metrics.

V. RESULTS

A. TESTING TOOL RESULTS

The three suites of testing tools provided interesting results on the relative quality of the three versions of the software under test. LDRA's Testbed reported an increasing number of programming-standard violations as the procedural version was first converted to the functional design then translated into the object-oriented version. The number of standards violations also increased as the design moved away from procedural design. Although the quantity of violations and the quantity of types of violations increased, the cyclomatic complexity decreased at each increment. Statement and branch coverage did not significantly differ across the three versions. While the other tools reported different information, their results were consistent with an increasing number of errors discovered in the non-procedural version yet increased levels of quality. Table 3 summarizes the findings.

Table 3. Summary of Tool Findings

	Procedural	**Functional**	**Object-Oriented**
LDRA	88% statement coverage	90% statement coverage	Not available
	83% branch coverage	86% branch coverage	Not available
	130 knots	109 knots	56 knots
	61 cyclomatic complexity	55 cyclomatic complexity	47 cyclomatic complexity
	109 occurrences of 18 different mandatory standards	115 occurrences of 18 different mandatory standards	401 occurrences of 31 different mandatory standards
	11 occurrences of 3 different checking standards	14 occurrences of 4 different checking standards	102 occurrences of 9 different checking standards
	80 occurrences of 6 different optional standards	36 occurrences of 6 different optional standards	75 occurrences of 9 different optional standards
Parasoft	95 occurrences of 8 different rules violations	83 occurrences of 8 different rules violations	122 occurrences of 12 different rules violations
Telelogic	218 occurrences of 14 different rules violations	Not available	372 occurrences of 20 different rules violations

The tools offer differing views of the quality of the software under test. When testing the procedural program, LDRA reported 200 occurrences of twenty-seven different coding standards, Telelogic reported a similar 218 occurrences but of only fourteen different rule violations, and Parasoft reported only ninety-five occurrences of only eight different rule violations. These differences can be attributed to the different standards and rules that are tested for by each tool. LDRA appends several industrial standards such as the Motor Industry Software Reliability Association (MISRA) C Standard and the Federal Aviation Authority's DO-178B standard. Likewise, the set of standards tested for by Telelogic and Parasoft intersect but are not identical.

Similar results occur when comparing tool results for the functional and object-oriented versions. Project managers should compare these differences to determine whether they would have an affect on the tool selection decision. If the additional standards used by LDRA are not an issue for current or prospective customers, the impact would be minimal.

B. TESTING TOOL METRIC RESULTS

After developing the proposed testing tool metrics, we applied them to the three testing-tool suites. During the process of applying the metrics, we discovered that several of the metrics are quite difficult, if not impossible, to calculate without having additional information supplied by the tool vendor. For example, if a vendor has not conducted a study on the tool's operational retainability by its users, experiments would need to be designed and conducted to evaluate the performance of users in applying the tools. If a vendor does not have statistics on its average response time to customer support requests, calculating the measure would be impossible.

Success was achieved in applying several of the metrics including human-interface design (HID), test-case generation (TCG), and reporting features (RF). HID measurements were calculated for each testing tool based on the sub-metrics of average keyboard-to-mouse switches (KMS), average input fields per function (IFPF), average length of input fields (ALIF), and button recognition (BR) when applicable. The sub-metrics demonstrated non-coarseness (different values were measured), finiteness (no metric was the same for all tools), and non-uniqueness (some equal values were obtained). The HID measurements were all unique, indicating that the measurement could be useful in comparing tools during the evaluation and selection process.

Test-case generation (TCG) measurements also provided unique measurements for each tool. Sub-metrics measuring levels of automated test-case generation (ATG) and test case reuse functionality (TRF) demonstrated the qualities of non-coarseness, finiteness, and non-uniqueness.

Reporting features (RF) measurements were also successful. It is simple to determine whether a tool automatically generates summary reports (SR) that are viewable without the tool application running (e.g., HTML or ASCII text document) (ER). The RF metric is non-coarse, finite, and non-unique. However, because each tool earned a SR score of one, additional testing should be conducted to determine SR's level of non-uniqueness.

Response time (RT) measurements for the three tools were all different. This indicates that RT is non-coarse and finite. Although not proven, it seems apparent that if two tools were to complete a test run in the same amount time, then they would receive a non-unique score.

No tools shared the same feature support (FS) nor Object-Oriented Software Quality Metrics (OOSWM) measurements. Therefore, they are non-coarse and finite, but an expanded study group of tools is required to verify their non-uniqueness.

Two tools earned the same metric-suite-supported score indicating non-uniqueness, while the third earned a different score showing the metric's non-coarseness and finiteness.

All three tools earned the same score in the Tool Management and User Control metrics. Further research must be conducted to determine the validity and usefulness of this metric.

The Maturity & Customer Base, Ease of Use, Tool Support, Estimated Return on Investment, Reliability, and Maximum Number of Parameters metrics were not completed. In order to do so would involve conducting extensive experiments or obtaining tool-vendor input, the latter of which is not readily available.

VI. CONCLUSION

A. SUMMARY

Automated testing tools vary in their underlying approach, quality, and ease-of-use, among other characteristics. Therefore, evaluating available tools and selecting the most appropriate suite of tools is vital to project success. The tool selection process, however, can be difficult and time-consuming due to the lack of metrics for measuring a tool's characteristics and comparing them to other tools. We have proposed a suite of objective metrics for measuring tool characteristics, to aid decision maker in systematically evaluating and selecting automated testing tools. These metrics are not tied to a specific architectural framework or programming language.

B. RECOMMENDATIONS FOR FUTURE WORK

The recommendations for future work include conducting more intensive testing of the tools cited in this thesis to include regression testing, testing the tools against larger and more complex software systems, including additional tools in the analyses, and analyzing the validity and utility of the proposed testing-tool metrics.

Conducting more intensive testing of previous tools includes creating additional test cases and modifying default test settings to improve test coverage and conducting regression testing. In our research, we used the default test settings of each tool to provide a baseline for measuring tool characteristics. Further research could be conducted to compare the testing tools under various operating system configurations and tool settings. Additional research could be conducted to measure a tool's capability and efficiency in measuring and improving testing coverage through modifying default settings and incorporating additional test cases. Research may also be conducted to measure a tool's ability to conduct and manage regression testing.

In our research, we implemented only three suites of testing tools that support C++ programs. Further work could incorporate a larger number of suites from different vendors such that a wide spectrum of programming languages are supported by the pool of tool suites. This would reduce the likelihood of language-specific factors affecting the research findings.

Our case study of a discrete-event simulation program could be supplemented by case studies for which the target software has a higher degree of encapsulation, inheritance, and polymorphism. The results could be used to determine the tools' capability and efficiency in dealing with such attributes.

The greatest opportunity for follow-on work is to research the validity and utility of the proposed suite of testing-tool metrics. The proposed metrics are based on the research conducted on evaluation versions of three commercially available suites of testing tools. Further research could be conducted to determine the metrics' validity in measuring the characteristics of testing tools and to ascertain their utility in evaluating and selecting among testing tools. For instance, are the metrics invalid and therefore useless? Do the proposed metrics provide valid measurements but provide minimal usefulness in the tool selection process? Or do they provide valid measurements that are useful in evaluating similarities and differences between automated software testing tools?

APPENDIX A. PROCEDURAL CODE

This appendix contains the procedural version of the CSMA/CD simulation program.

```
# include <stdio.h>

# include <stdlib.h>

# include <math.h>


# define    MAX_STATIONS         10      /* Number of stations */

# define    BUS_RATE        2000000.0    /* Transmission rate in bps*/

# define    PACKET_LENGTH      1000.0    /* Packet length (bits) */

# define    BUS_LENGTH         2000.0    /* Bus length in meters */

# define    MAX_BACKOFF          15.0    /* Backoff period in slots*/

# define    PERSIST               0.0    /* Persistence */

# define    JAM_PERIOD            5.0    /* Jamming period */

# define     MAX_PACKETS          226        /* Maximum packets to be
transmitted in a simulation run */

# define    FACTOR              1000.0    /* A factor used for changing
units of time */

# define    MAX_Q_SIZE          50      /* Maximum queue size */

# define    ID_SIZE            50      /* Size of the identity array */

# define    DEGREES_FR            5      /* Degrees of freedom */


double arrival_rate; /* arrival rate (in packets/sec) per station */

double  arrival_rate_slots;  /*  arrival  rate  (in  packets/slot)  per
station */

double packet_time; /* packet transmission time */
double                                                    t_dist_par[10]
={12.706,4.303,3.182,2.776,2.571,2.447,2.365,2.306,
                                              2.262,  2.228};  /*
T-distribution parameters */
double start_time [ID_SIZE]; /* starting time of packet */
double event_time [MAX_STATIONS][4]; /* time of occurrence of an event
*/
double delay_ci [DEGREES_FR + 1]; /* array to store delay values */
double utilization_ci[DEGREES_FR + 1]; /* array to store utilization
values*/
double throughput_ci[DEGREES_FR + 1]; /* array to store throughput
values */
double collision_rate_ci[DEGREES_FR + 1]; /* array for collision rate
values*/

double slot_size, p, ch_busy;
double rho, clock, d_clock, no_pkts_departed, next_event_time;
double x, logx, rand_size, infinite;
double delay, total_delay, average_delay;
double delay_sum, delay_sqr, delay_var, delay_sdv, delay_con_int;
double utilization, utilization_sum, utilization_sqr;
double utilization_var, utilization_sdv, utilization_con_int;
double throughput, throughput_sum;
double collision_rate, collision_rate_sum, collision_end_time;
double select_prob, backoff_time, packet_slots;

int queue_size [MAX_STATIONS]; /* current queue size at a station */
int queue_id [MAX_STATIONS][MAX_Q_SIZE]; /* array for id_numbers of
packets */
int id_list [ID_SIZE]; /* array of id_numbers */
int id_attempt_stn [MAX_STATIONS]; /* array to identify attempting
stations */

int  j, ic, ii, next_station, next_event, next, id_number;
int no_attempts, no_trans, no_collisions, select_flag;

int main ()
{
printf("The following results are for: \n");
printf("Degrees of freedom = %d\n", DEGREES_FR);
printf("Confidence Interval = 95 percent \n");
printf("========================================= \n");
printf("\n");

arrival_rate =  0.0;
slot_size    =  BUS_LENGTH * FACTOR * 5.0 * pow (10.0, -9.0);
p            =  PERSIST;
packet_time  =  PACKET_LENGTH * FACTOR / BUS_RATE;
packet_slots =  (double) (int) (packet_time/slot_size) + 1.0;
infinite     =  1.0 * pow (10.0, 30.0);
rand_size = RAND_MAX;

for (ii=0; ii < 10; ii++)
  {
  arrival_rate = arrival_rate + 20.0;

  for (ic = 0; ic <= DEGREES_FR; ic++)
    {
```

```
rho                   = 0.0;

ch_busy               = 0.0;

clock                 = 0.0;

d_clock               = 0.0;

collision_end_time    = 0.0;

utilization           = 0.0;

no_pkts_departed      = 0.0;

total_delay           = 0.0;

next_event_time       = 0.0;

average_delay         = 0.0;

no_collisions         = 0;

select_flag           = 0;


/* Compute the traffic intensity. If the traffic intensity

   is greater than unity, stop the program. */


rho = arrival_rate * PACKET_LENGTH * MAX_STATIONS / BUS_RATE;


if (rho >= 1.0)

  {

  printf("Traffic intensity is too high\n");

  exit(1);

  }


/* Initialize all variables to their appropriate values. */


arrival_rate_slots = arrival_rate * slot_size;

for (int i = 0; i < MAX_STATIONS; i++) queue_size[i]=0;
for (int k = 0; k < ID_SIZE; k++)
  {
  start_time[k] = 0.0;
  id_list[k] = 0;
  }

for (int m = 0; m < MAX_STATIONS; m++)
  {
  for(int l = 0; l < MAX_Q_SIZE; l++) queue_id[m][l]=0;
  }

for (int n = 0; n < MAX_STATIONS; n++)
  {
  for (j = 0; j < 4; j++)
    {
    event_time[n][j] = infinite;
    x = (double) rand();
    x = x * FACTOR/rand_size;
    if (j == 0) event_time[n][j] = x;
    }
  }

/* Scan the event list and pick the next event to be executed. */

while (no_pkts_departed < MAX_PACKETS)
  {
  next_event_time = infinite;
  for (int i = 0; i < MAX_STATIONS; i++)
    {
    for (j = 0; j < 4; j++)
      {
      if (next_event_time > event_time[i][j])
        {
        next_event_time = event_time[i][j];
        next_station = i;
        next_event   = j;
        }
      }
    }
  clock = next_event_time;

  if (next_event > 3)
    {
    printf("Check the event-list");
    exit(1);
    }
  while (d_clock <= clock) d_clock ++ ;
  switch (next_event)
    {
    case 0:   /* This is an arrival event. */
      {

      /* Select an identification for the arriving message */

      id_number = -1;
```

```
          for (int i = 0; i < ID_SIZE; i++)
            {
            if (id_list[i] == 0)
              {
              id_number = i;
              id_list[i] = 1;
              break;
              }
            if (id_number != -1) continue;
            }



         if (id_number == -1)
            {
            printf("Check the ID-list.");
            exit(1);
            }
          queue_size[next_station] ++ ;
          if (queue_size[next_station] > MAX_Q_SIZE)
            {
            printf("The   queue   size   is   large   and   is   =   %d\n",
queue_size[next_station]);
            exit(1);
            }
          queue_id[next_station][(queue_size[next_station]-1)]         =
id_number;
          start_time[id_number] = clock;
```

```
  if (queue_size[next_station] == 1)
    {
    event_time[next_station][1] = d_clock;
    if (event_time[next_station][1] <= collision_end_time)
       event_time[next_station][1] = collision_end_time + 1.0;
    }

  /* Schedule the next arrival */

  for (;;)
    {
    x = (double) rand();
    if (x != 0.0) break;
    }
  logx = -log(x/rand_size) * FACTOR / arrival_rate_slots;
  event_time[next_station][next_event] = clock + logx;
  break;
  }

case 1:   /* This is an attempt event. */
  {
  no_attempts = 0;
  for (int i = 0; i < MAX_STATIONS; i++)
    {
    if (event_time[i][1] == clock)
      {
      no_attempts ++ ;
      id_attempt_stn[no_attempts - 1] = i;
      }
    }
  select_flag = 0;
  if (no_attempts > 1)
    {
    x = (double) rand();
    x = x/rand_size;
    for (int i = 0; i < no_attempts; i++)
      {
      select_prob = (double) (i+1)/ ((double) no_attempts);
      if (x <= select_prob)
        {
        next_station = id_attempt_stn[i];
        select_flag = 1;
        }
      if (select_flag == 1) continue;
      }
    }
  if (ch_busy == 0.0)
    {
    if (p == 0.0)
      {
      event_time[next_station][2] = clock + 1.0;
      event_time[next_station][1] = infinite;
      }
    else
      {
      x = (double) rand();
```

```
            x = x/rand_size;
            if (x < p)
              {
              event_time[next_station][2] = clock + 1.0;
              event_time[next_station][1] = infinite;
              }
            else
              {
              event_time[next_station][1] = clock + 1.0;
              if (event_time[next_station][1] <= collision_end_time)
              event_time[next_station][1] = collision_end_time + 1.0;
              event_time[next_station][2] = infinite;
              }
            }
          }

        if (ch_busy == 1.0)
          {
          if (p == 0.0)
            {
            x = (double) rand();
            x = x/rand_size;
            backoff_time = (double) (int) (x * MAX_BACKOFF);
            if (backoff_time < 1.0) backoff_time = 1.0;
            event_time[next_station][1] = clock + backoff_time;
            if (event_time[next_station][1] <= collision_end_time)
            event_time[next_station][1]    =    collision_end_time    +
backoff_time;
```

```
      event_time[next_station][2] = infinite;
      }
    else
      {
      event_time[next_station][1] = clock + 1.0;
      if (event_time[next_station][1] <= collision_end_time)
      event_time[next_station][1] = collision_end_time + 1.0;
      event_time[next_station][2] = infinite;
      }
    }
  break;
  }
case 2: /* This is a transmission event */
  {
  no_trans = 0;
  for (int i = 0; i < MAX_STATIONS; i++)
  if (event_time[i][2] == clock) no_trans ++ ;
  if (no_trans > 1)
    {
      {
      collision_end_time = clock + JAM_PERIOD + 2.0;
      no_collisions ++ ;
      }
    for (int i = 0; i < MAX_STATIONS; i++)
      {
      if (event_time[i][2] == clock)
        {
        event_time[i][2] = infinite;
        x = (double) rand();
        x = x/rand_size;
        backoff_time = (double) (int) (x * MAX_BACKOFF);
        if (backoff_time < 1.0) backoff_time = 1.0;
        event_time[i][1] = collision_end_time + backoff_time;
        }
      if (event_time[i][1] <= collision_end_time)
        {
        x = (double) rand();
        x = x/rand_size;
        backoff_time = (double) (int) (x * MAX_BACKOFF);
        if (backoff_time < 1.0) backoff_time = 1.0;
        event_time[i][1] = collision_end_time + backoff_time;
        }
      }
    }
  else
    {
    if (ch_busy != 1.0)
      {
      event_time[next_station][3] = clock + packet_slots ;
      event_time[next_station][2] = infinite;
      ch_busy = 1.0;
      }
    else
      {
      if (p == 0.0)
        {
                x = (double) rand();

                x = x/rand_size;

                backoff_time = (double) (int) (x * MAX_BACKOFF);

                if (backoff_time < 1.0) backoff_time = 1.0;

                event_time[next_station][1] = clock + backoff_time;

                if (event_time[next_station][1] <= collision_end_time)

                event_time[next_station][1]   =   collision_end_time   +
backoff_time;

                event_time[next_station][2] = infinite;

                }

              else

                {

                event_time[next_station][1] = clock + 1.0;

                if (event_time[next_station][1] <= collision_end_time)

                event_time[next_station][1] = collision_end_time + 1.0;

                event_time[next_station][2] = infinite;

                }

              }

            }

          break;

          }


        case 3:   /* This is a departure event */

          {

          id_number = queue_id[next_station][0];

          ch_busy = 0.0;

          queue_size[next_station] -- ;
```

```
      /* Push the queue forward */

      for (int i = 0; i < queue_size[next_station]; i++)
        queue_id[next_station][i] = queue_id[next_station][i+1];
      queue_id[next_station][queue_size[next_station]] = 0;
      delay = clock - start_time[id_number];
      total_delay += delay;
      id_list[id_number] = 0;
      no_pkts_departed += 1.0;
      utilization += packet_slots;
      event_time[next_station][3] = infinite;
      if (queue_size[next_station] > 0)
        {
        event_time[next_station][1] = clock + 1.0;
        if (event_time[next_station][1] <= collision_end_time)
        event_time[next_station][1] = collision_end_time + 1.0;
        }
      else
        {
        event_time[next_station][1] = infinite;
        event_time[next_station][2] = infinite;
        }
      break;
      }
    }
  }
utilization = utilization / clock;
```

```
    average_delay = total_delay * slot_size / (no_pkts_departed *
FACTOR);
    throughput = no_pkts_departed * FACTOR / (clock * slot_size);
    collision_rate = (double) no_collisions * FACTOR / (clock *
slot_size);
    utilization_ci[ic] = utilization;
    delay_ci[ic] = average_delay;
    throughput_ci[ic] = throughput;
    collision_rate_ci[ic] = collision_rate;
    }
  delay_sum = 0.0;
  delay_sqr = 0.0;
  utilization_sum = 0.0;
  utilization_sqr = 0.0;
  throughput_sum = 0.0;
  collision_rate_sum = 0.0;
  for (ic = 0; ic <= DEGREES_FR; ic++)
    {
    delay_sum += delay_ci[ic];
    delay_sqr += pow (delay_ci[ic],2.0);
    utilization_sum += utilization_ci[ic];
    utilization_sqr += pow (utilization_ci[ic],2.0);
    throughput_sum += throughput_ci[ic];
    collision_rate_sum += collision_rate_ci[ic];
    }
  delay_sum = delay_sum / (DEGREES_FR + 1);
  delay_sqr = delay_sqr / (DEGREES_FR + 1);
  delay_var = delay_sqr - pow(delay_sum,2.0);
  delay_sdv = sqrt(delay_var);
  delay_con_int      =      delay_sdv      *      t_dist_par[DEGREES_FR-
1]/sqrt(DEGREES_FR);
  utilization_sum = utilization_sum / (DEGREES_FR + 1);
  utilization_sqr = utilization_sqr / (DEGREES_FR + 1);
  utilization_var = utilization_sqr - pow(utilization_sum,2.0);
  utilization_sdv = sqrt(utilization_var);
  utilization_con_int   =   utilization_sdv   *   t_dist_par[DEGREES_FR-
1]/sqrt(DEGREES_FR);
  throughput_sum = throughput_sum / (DEGREES_FR + 1);
  collision_rate_sum = collision_rate_sum / (DEGREES_FR + 1);


  printf("For an arrival rate = %g\n",arrival_rate);
  printf("The traffic intensity = %g\n", rho);
  printf("The average delay = %g", delay_sum);
  printf(" +- %g\n", delay_con_int);
  printf("The utilization = %g", utilization_sum);
  printf(" +- %g\n", utilization_con_int);
  printf("The throughput = %g\n", throughput_sum);
  printf("The collision rate = %g\n", collision_rate_sum);
  printf("\n");
  }

return 0;
}
```

APPENDIX B. FUNCTIONAL CODE

This appendix contains the functional version of the CSMA/CD simulation program. It was created by the authors through the process of implementing functions to replace sections of code that appeared several times in the procedural version in Appendix A.

```
# include <stdio.h>

# include <stdlib.h>

# include <math.h>


# define MAX_STATIONS         10    /* Number of stations */

# define BUS_RATE       2000000.0  /* Transmission rate in bps*/

# define PACKET_LENGTH     1000.0  /* Packet length (bits) */

# define BUS_LENGTH        2000.0  /* Bus length in meters */

# define MAX_BACKOFF         15.0  /* Backoff period in slots*/

# define PERSIST              0.0  /* Persistence */

# define JAM_PERIOD           5.0  /* Jamming period */

# define MAX_PACKETS          10    /* Maximum packets to be transmitted
in a simulation run */

# define FACTOR             1000.0  /* A factor used for changing units
of time */

# define MAX_Q_SIZE         500    /* Maximum queue size */

# define ID_SIZE             50    /* Size of the identity array */

# define DEGREES_FR           5    /* Degrees of freedom */


//arrays

double t_dist_par[10] ={12.706, 4.303, 3.182, 2.776, 2.571,
                        2.447,  2.365,  2.306,  2.262,  2.228};  /*  T-
distribution parameters */
double start_time [ID_SIZE];            /* starting time of packet */
double  event_time  [MAX_STATIONS][4];      /*  time  of  occurrence  of  an
event */
double delay_ci [DEGREES_FR + 1];        /* array to store delay values
*/
double utilization_ci[DEGREES_FR + 1];     /* array to store utilization
values*/
double throughput_ci  [DEGREES_FR + 1];     /* array to store throughput
values */
double  collision_rate_ci[DEGREES_FR  +  1];  /*  array  for  collision  rate
values*/


//numbers
double arrival_rate;        /* arrival rate (in packets/sec) per station
*/
double   arrival_rate_slots;  /*   arrival   rate   (in   packets/slot)   per
station */
double packet_time;        /* packet transmission time */


double slot_size,
      persistence,
      ch_busy;
double rho,
      clock,
      d_clock,
      num_pkts_departed,
      next_event_time;
double x,
      logx,

      rand_size,

      infinite;

double delay,

      total_delay,

      average_delay;

double delay_sum,

      delay_sqr,

      delay_var,

      delay_sdv,

      delay_con_int;

double utilization,

      utilization_sum,

      utilization_sqr;

double utilization_var,

      utilization_sdv,

      utilization_con_int;

double throughput,

      throughput_sum;

double collision_rate,

      collision_rate_sum,

      collision_end_time;

double select_prob,

      backoff_time,

      packet_slots;


int queue_size [MAX_STATIONS];             /* current queue size at a
station */
```

```
int  queue_id  [MAX_STATIONS][MAX_Q_SIZE];  /*  array  for  id_numbers  of
packets */

int id_list [ID_SIZE];                   /* array of id_numbers */

int  id_attempt_stn  [MAX_STATIONS];             /*  array  to  identify
attempting stations */


//First ints (i, j, ic, ii) only used as local for loop counters.

//They didn't need to be globals.

int next_station,

    next_event,

    id_number;

int num_attempts,

    num_trans,

    num_collisions,

    select_flag;


//-----------------------------------------------------------------------
---

//Function:  pick_event

//Purpose:   Scan event list looking for station-event pair with

//           the earliest time.  Clock moved to this time.

//Post:      List is not changed.

//-----------------------------------------------------------------------
---

void pick_event ()

{

   //set next event time infinitely large

   next_event_time = infinite;
   //scan all stations'...

   for (int i = 0; i < MAX_STATIONS; i++)

   {

      //queues

      for (int j = 0; j < 4; j++)

      {

         //if next event time is after station's event's time

         if (next_event_time > event_time[i][j])

         {

            //set next event time to station's event's time

            next_event_time = event_time[i][j];


            //set station and event of next event to perform

            next_station = i;

            next_event   = j;

         }//end of if


      }//end of j for


   }//end of i for


   //after finding next soonest event to be performed, set clock

   //to that event time

   clock = next_event_time;


   return;


};//end of pick_event()
```

```
//----------------------------------------------------------------------
---
//Function:  arrival_event
//Purpose:    Simulate the arrival of a packet at a station that needs
to
//           be transmitted.
//Post:      Packet is given an id number (if available).
//           Station's queue size is incremented if not full.
//           Packet id is placed in queue_id array.
//           If new packet is station's only packet, schedule time to
//           attempt transmission.
//           Packet is given start time.
//----------------------------------------------------------------------
---
void arrival_event ()
{
   /* Select an identification for the arriving message */

   //set identification number to -1
   id_number = -1;

   //for all i less than number of ids
   for (int i = 0; i < ID_SIZE; i++)
   {
      //if 'flag' at i is zero
      if (id_list[i] == 0)
      {
```

```
      //set id number of packet to i
      id_number = i;
      //set 'flag' at i to one
      id_list[i] = 1;
      break;
   }

   //if id number was available continue
   if (id_number != -1) continue;
}

//if id number wasn't changed, either id's are gone or there
//is a problem with the id list...
if (id_number == -1)
{
   //inform user to check id list and stop program
   printf("Check the ID-list.");
   exit(1);
}

//increment station's queue size (to hold new packet)
queue_size[next_station] ++ ;

//if station's queue size is too big...
if (queue_size[next_station] > MAX_Q_SIZE)
{
   //inform user and stop program
```

```
      printf("The    queue    size    is    large    and    is    =    %d\n",
queue_size[next_station]);
      exit(1);
   }


   //place packet in station's queue
   queue_id[next_station][(queue_size[next_station]-1)] = id_number;


   //set packet's start time to current clock value
   start_time[id_number] = clock;


   //if new packet is the station's only packet..
   if (queue_size[next_station] == 1)
   {
      //schedule transmission attempt
      event_time[next_station][1] = d_clock;


      //if a collision period is active...
      if (event_time[next_station][1] <= collision_end_time)
      {
         //delay transmission attempt til collision ends
         event_time[next_station][1] = collision_end_time + 1.0;
      }
   }



   //Schedule the next arrival for the current station
   //this is a version of a do/while loop
   for (;;)
   {
      //find a random number that's not zero
      x = (double) rand();
      if (x != 0.0) break;
   }

   //use x to schedule next arrival for current station
   logx = -log(x/rand_size) * FACTOR / arrival_rate_slots;
   event_time[next_station][next_event] = clock + logx;

   return;

};//end of arrival_event ()



//----------------------------------------------------------------------
---
//Function:  find_backoff_time()
//Purpose:    Generate a random number which is then used to pick a
random
//           backoff time for rescheduling a transmission attempt.
//Post:      No changes to any packet times is made.
//----------------------------------------------------------------------
---
void find_backoff_time()
{
   //generate a random number to..
   x = (double) rand();

   x = x/rand_size;

   //calculate backoff time

   backoff_time = (double) (int) (x * MAX_BACKOFF);

   //if backoff time is < 1, set to 1.0

   if (backoff_time < 1.0) backoff_time = 1.0;



   return;

}; //end of find_backoff_time()




//----------------------------------------------------------------------
---

//Function:  resched_attempt()

//Purpose:   If zero persistence, reschedule attempt for random time in

//           in the future.  If one persistence, reschedule attempt for

//           next available time.

//Post:           Packet  attempt  time  is  given  value  determined  by
persistence.

//           Packet transmission time is reset to infinite.

//----------------------------------------------------------------------
---

void resched_attempt ()

{

   //if persistence is zero...

   if (persistence == 0.0)

   {

      //call function to find backoff time

      find_backoff_time();
      //set attempt time to current time plus backoff time

      event_time[next_station][1] = clock + backoff_time;

      //if attempt time falls in a collision period...

      if (event_time[next_station][1] <= collision_end_time)

      //set attempt time to collision end plus backoff time

      event_time[next_station][1] = collision_end_time + backoff_time;

      //set event transmission time to infinite

      event_time[next_station][2] = infinite;

   }

   else//if persistence is not zero

   {

      //set attempt time to next clock increment

      event_time[next_station][1] = clock + 1.0;

      //if new attempt time falls in collision period...

      if (event_time[next_station][1] <= collision_end_time)

      //set attempt time to one clock increment past collision

      event_time[next_station][1] = collision_end_time + 1.0;

      //reset event transmission time

      event_time[next_station][2] = infinite;

   }


}; //end of resched_attempt()


//----------------------------------------------------------------------
---

//Function:  attempt_event

//Purpose:   Determine number of stations trying to transmit.

//           If more than 1, pick one at random.
//           If channel is not busy, and..
//             zero persistence: transmit packet
//             p persistence:    transmit randomly
//           If channel is busy,
//             zero persistence: find random backoff time and resched
//             p persistence:    resched for next available slot
//
//
//Post:      Packet attempt and transmit times will be altered based
//           on nested if/else statements.
//---------------------------------------------------------------------
---
void attempt_event ()
{
   //set number of attempts to zero
   num_attempts = 0;

   //for all stations...
   for (int i = 0; i < MAX_STATIONS; i++)
   {
      //if station's next event is ready to transmit
      if (event_time[i][1] == clock)
      {
         //increment number of stations attempting to transmit
         num_attempts ++ ;

         //place id of attempting station in an array
         id_attempt_stn[num_attempts - 1] = i;
      }
   }
   //set select flag to zero
   select_flag = 0;

   //if more than one station is trying to transmit...
   if (num_attempts > 1)
   {
      //pick a random number
      x = (double) rand();
      x = x/rand_size;

      //for all stations trying to trasmit...
      for (int i = 0; i < num_attempts; i++)
      {
         //determine select prob
         select_prob = (double) (i+1)/ ((double) num_attempts);

         //if random number is less than select prob...
         if (x <= select_prob)
         {
            //then station is slot i of array gets to transmit
            next_station = id_attempt_stn[i];
            //set selection flag to 'true'
            select_flag = 1;
         }

         //if a station has been chosen continue
         if (select_flag == 1) continue;
      }
   }

   //if channel is not busy...
   if (ch_busy == 0.0)
   {
      //and if persistence is zero
      if (persistence == 0.0)
      {
         //transmission time of current event is set
         event_time[next_station][2] = clock + 1.0;
         //attempt time of current event is reset
         event_time[next_station][1] = infinite;
      }
      else//if persistence is not zero
      {
         //generate a random number
         x = (double) rand();
         x = x/rand_size;
         //if random number is less than persistence
         if (x < persistence)
         {
            //set transmission time of current event
            event_time[next_station][2] = clock + 1.0;
            //reset attempt time of current event
            event_time[next_station][1] = infinite;
         }
```

```
         else//random number is greater than persistence
         {
            //set attempt time of current event (event not transmitted)
            event_time[next_station][1] = clock + 1.0;
            //if new attempt time falls in a collision time...
            if (event_time[next_station][1] <= collision_end_time)
               //delay next attempt til collision ends
               event_time[next_station][1] = collision_end_time + 1.0;
            //ensure transmission time of current event is reset
            event_time[next_station][2] = infinite;
         }
      }
   }

   //if channel is busy...
   if (ch_busy == 1.0)

      //reschedule transmission attempt time
      resched_attempt();



   return;

};//end of attempt_event()

//----------------------------------------------------------------------
---
//Function:  transmit_event
```

```
//Purpose:   Determine number of statinos trying to transmit.
//           If more than one, there is a collision; jam for specified
//               time period and have all transmitting stations resched
//               attempt to transmit.
//           If only one, and
//              channel not busy, transmit.
//              channel busy, determine reschedule time.
//Post:      Packet attempt/transmit time(s) will be updated.
//-----------------------------------------------------------------------
---
void transmit_event ()
{
   //set number of transmission to zero
   num_trans = 0;
   //for all stations...
   for (int i = 0; i < MAX_STATIONS; i++)
   {
      //if  they  are  currently  transmitting,  increment  number  of
transmissions
      if (event_time[i][2] == clock) num_trans ++ ;


      //if there are more than one transmitting stations,
      //there's a collision and ...
      if (num_trans > 1)
      {
         //set collision end time
         collision_end_time = clock + JAM_PERIOD + 2.0;
         //increment collision count
         num_collisions ++ ;
         //for all stations...
         for (int i = 0; i < MAX_STATIONS; i++)
         {
            //if they are transmitting...
            if (event_time[i][2] == clock)
            {
               //set transmit time to infinite
               event_time[i][2] = infinite;

               //call function to find backoff time
               find_backoff_time();
               //set  attempt  time  to  collision  end  time  plus  backoff
time
               event_time[i][1] = collision_end_time + backoff_time;
            }

            //if attempt time is before collision end time
            if (event_time[i][1] <= collision_end_time)
            {
               //call function to find backoff time
               find_backoff_time();
               //set  attempt  time  to  collision  end  time  plus  backoff
time
               event_time[i][1] = collision_end_time + backoff_time;
            }
         }
      }
      else//only one channel trying to transmit
      {
         //if channel not busy...
         if (ch_busy != 1.0)
         {
            //set departure time to clock time + # of packet slots sent
            event_time[next_station][3] = clock + packet_slots ;
            //reset event transmission time
            event_time[next_station][2] = infinite;
            //set channel to busy cause transmitting
            ch_busy = 1.0;
         }
         else//if channel is busy..
         {
            //reschedule transmission attempt time
            resched_attempt();

         }

      }
   }
   return;

};//end of transmit_event ()

//-----------------------------------------------------------------------
---
//Function:  departure_event
//Purpose:   Simulate successful transmission.
```

```
//Post:      Free channel and update transmitting station's queue size.
//                   Calculate  delay  of  packet  and  other  performance
parameters.
//                If  station's  queue  not  empty,  schedule  next  transmit
attempt.
//----------------------------------------------------------------------
---
void departure_event ()
{
   //let id_number be identification number of departing packet
   id_number = queue_id[next_station][0];

   //free transmission medium
   ch_busy = 0.0;

   //decrement queue size
   queue_size[next_station] -- ;

   //Push the queue forward
   for (int i = 0; i < queue_size[next_station]; i++)
            queue_id[next_station][i] = queue_id[next_station][i+1];

   //set departing packet id to 0
   queue_id[next_station][queue_size[next_station]] = 0;

   //calculate delay for departing packet
   delay = clock - start_time[id_number];

   //add delay to total_delay
```

```
total_delay += delay;


//release id number of departing packet
id_list[id_number] = 0;


//increment number of packets departed
num_pkts_departed += 1.0;


//add packet slots to utilization
utilization += packet_slots;


//reset departing packet event time
event_time[next_station][3] = infinite;


//if station's queue is not empty
if (queue_size[next_station] > 0)
   {
   //schedule station's next event for transmission attempt
   event_time[next_station][1] = clock + 1.0;


      //if event time is prior to end of collision end time
      if (event_time[next_station][1] <= collision_end_time)
         //set event time to one increment after collision end time
         event_time[next_station][1] = collision_end_time + 1.0;
      }
else //if station's queue is empty
   {
   //reset event attempt and transmission times
```

```
      event_time[next_station][1] = infinite;

      event_time[next_station][2] = infinite;

      }


   return;

}//end of departure_event




//----------------------------------------------------------------------
---

//Function:  calculate_results

//Purpose:   Calculate performance statistics.

//Post:      Lists are not changed.

//----------------------------------------------------------------------
---

void calculate_results ()

{

   //initialize values

   delay_sum = 0.0;

   delay_sqr = 0.0;

   utilization_sum = 0.0;

   utilization_sqr = 0.0;

   throughput_sum = 0.0;

   collision_rate_sum = 0.0;


   //calculate confidence level values

   for (int ic2 = 0; ic2 <= DEGREES_FR; ic2++)

   {

      delay_sum += delay_ci[ic2];
      delay_sqr += pow (delay_ci[ic2],2.0);

      utilization_sum += utilization_ci[ic2];

      utilization_sqr += pow (utilization_ci[ic2],2.0);

      throughput_sum += throughput_ci[ic2];

      collision_rate_sum += collision_rate_ci[ic2];

   }


   //calculate arrival rate's stats

   delay_sum = delay_sum / (DEGREES_FR + 1);

   delay_sqr = delay_sqr / (DEGREES_FR + 1);

   delay_var = delay_sqr - pow(delay_sum,2.0);

   delay_sdv = sqrt(delay_var);

   delay_con_int      =      delay_sdv      *      t_dist_par[DEGREES_FR-
1]/sqrt(DEGREES_FR);

   utilization_sum = utilization_sum / (DEGREES_FR + 1);

   utilization_sqr = utilization_sqr / (DEGREES_FR + 1);

   utilization_var = utilization_sqr - pow(utilization_sum,2.0);

   utilization_sdv = sqrt(utilization_var);

   utilization_con_int   =   utilization_sdv   *   t_dist_par[DEGREES_FR-
1]/sqrt(DEGREES_FR);

   throughput_sum = throughput_sum / (DEGREES_FR + 1);

   collision_rate_sum = collision_rate_sum / (DEGREES_FR + 1);


  return;


};//end of calculate_results
```

```
//----------------------------------------------------------------------
---
//Function:  show_results
//Purpose:   Display results of simulation.
//Post:      Lists are not changed.
//----------------------------------------------------------------------
---
void show_results ()
{
   //display statistics for specified arrival rate
   printf("For an arrival rate = %g\n",arrival_rate);
   printf("The traffic intensity = %g\n", rho);
   printf("The average delay = %g", delay_sum);
   printf(" +- %g\n", delay_con_int);
   printf("The utilization = %g", utilization_sum);
   printf(" +- %g\n", utilization_con_int);
   printf("The throughput = %g\n", throughput_sum);
   printf("The collision rate = %g\n", collision_rate_sum);
   printf("\n");

   return;

};//end of show_results ()



//----------------------------------------------------------------------
//Beginning of main function.
int main ()
{
   //Print intro

   printf("The following results are for: \n");

   printf("Degrees of freedom = %d\n", DEGREES_FR);

   printf("Confidence Interval = 95 percent \n");

   printf("========================================= \n");

   printf("\n");


   //set/calculate some initial values

   arrival_rate =  0.0;

   slot_size    =  BUS_LENGTH * FACTOR * 5.0 * pow (10.0, -9.0);

   persistence  =  PERSIST;

   packet_time  =  PACKET_LENGTH * FACTOR / BUS_RATE;

   packet_slots =  (double) (int) (packet_time/slot_size) + 1.0;

   infinite     =  1.0 * pow (10.0, 30.0);

   rand_size    =  RAND_MAX;


   //perform simulation several times...

   for (int ii=0; ii < 10; ii++)

   {

      //incrementing arrival rate by 20 units each pass through

      arrival_rate = arrival_rate + 20.0;


      //each arrival rate is simulated several times to find confidence
levels

      for (int ic = 0; ic <= DEGREES_FR; ic++)

      {

         rho                  = 0.0;  //traffic intensity (max value is
1.0)
```

```
         ch_busy              = 0.0;  //channel is free if 0, otherwise
channel is busy
         clock               = 0.0;  //simulation clock
         d_clock             = 0.0;  //clock to determine slot timing
         collision_end_time  = 0.0;  //the time at which collision ends
         utilization         = 0.0;  //utilization of channel
         num_pkts_departed   = 0.0;  //number of packets sent
         total_delay         = 0.0;  //aggregrate delay of packets
         next_event_time     = 0.0;  //time to execute next event
         average_delay           =  0.0;   //average  of  delays  for  all
packets
         num_collisions      = 0;    //number of collisions that occur
         select_flag             =  0;      //indicates  if  transmitting
station is selected

         //Compute the traffic intensity
         rho = arrival_rate * PACKET_LENGTH * MAX_STATIONS / BUS_RATE;

         //If greater than unity...
         if (rho >= 1.0)
         {
            //tell user and stop program
            printf("Traffic intensity is too high. \n");
            exit(1);
         }


         /* Initialize all variables to their appropriate values. */
         arrival_rate_slots = arrival_rate * slot_size;
```

```
//set queue size at all stations to zero
for (int i = 0; i < MAX_STATIONS; i++)
   queue_size[i]=0;

//set start time and id list of all stations to zero
for (int k = 0; k < ID_SIZE; k++)
{
   start_time[k] = 0.0;
   id_list[k] = 0;
}

//for all stations...
for (int m = 0; m < MAX_STATIONS; m++)
{
   //set all event ids to zero
   for(int n = 0; n < MAX_Q_SIZE; n++) queue_id[m][n]=0;
}

//for all stations...
for (int q = 0; q < MAX_STATIONS; q++)
{
   for (int r = 0; r < 4; r++)
   {
      //set event times to infinite
      event_time[q][r] = infinite;
      //get a random number
      x = (double) rand();
```

```
            x = x * FACTOR/rand_size;
            //set first event time to random value
            if (r == 0) event_time[q][r] = x;
         }//end of r for loop

      }//end of q loop

      //while have NOT exceeded the max # of packets to send...
      while (num_pkts_departed < MAX_PACKETS)
      {
         //scan event list for next event
         pick_event();

         //if get invalid event type...
         if (next_event > 3)
         {
            //inform user of problem with event list & stop program
            printf("Check the event-list");
            exit(1);
         }//end if

         //while  slot  clock  is  less  than  simulation  clock,  go  to
next slot
         while (d_clock <= clock) d_clock ++ ;

         //use type of next event to choose function to call
         switch (next_event)
         {
            //if an arrival event, call arrival function
            case 0:
               arrival_event ();
               break;

            //if an attempt_event, call attempt function
            case 1:
               attempt_event();
               break;

            //if a transmission event, call transmit_event
            case 2:
               transmit_event ();
               break;

            //if a packet departure. call departure function
            case 3:
               departure_event ();
               break;

         }//end of switch

      }//end of while  -- done with simulation pass

      //calculate statistics for pass
      utilization = utilization / clock;
      average_delay = total_delay * slot_size / (num_pkts_departed *
FACTOR);
```

```
         throughput = num_pkts_departed * FACTOR / (clock * slot_size);
         collision_rate = (double) num_collisions * FACTOR / (clock *
slot_size);


         //place results in array for calculating confidence factors
         utilization_ci[ic] = utilization;
         delay_ci[ic] = average_delay;
         throughput_ci[ic] = throughput;
         collision_rate_ci[ic] = collision_rate;

      }//end of ic for loop

      //find results for current arrival rate
      calculate_results();

      //display results for current arrival rate
      show_results();

   }//end of ii for loop

   return 0;

}//end of main
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. OBJECT-ORIENTED CODE

This appendix contains the object-oriented version of the CSMA/CD simulation program. This version was built on the work of Neil Acantilado, including his UML diagram shown in Appendix D.

The program is divided into five classes, a main file, and a file containing constants. The simulation.cpp file contains the C++ main function and includes all the other files. The constants.h file contains thirteen constants that are used throughout the project by various entities. The remaining ten files consist of the .cpp and .h files that create the five classes of objects used by the project to represent the network, stations on a network, a list of events that will occur, a list of ids that represent the packets of information on the network, and a clock for timing purposes.

C-1 SIMULATION.CPP

```
# include <stdio.h>

# include <stdlib.h>

# include <math.h>


#include "Constants.h"

#include "Clock.h"

#include "EventList.h"

#include "IdList.h"

#include "Network.h"

#include "stationList.h"



   // Holds a list of stations and their queues

   StationList stations;


   // Holds a list of id_numbers and their queues

   IdList ids;


   // Holds a list of 4 type of events

   EventList events;


   // Clock representing simulation process.

   Clock watch;


   // Keeps track of status of channel

   Network csmaNetwork(PERSIST);
```

```
// Arrival rate (in packets/sec) per station

float arrival_rate = 0.0;


// Represents the end-to-end propagation delay

float slot_size = BUS_LENGTH * FACTOR * 5.0 * pow (10.0, -9.0);


// Arrival rate (in packets/slot) per station

float arrival_rate_slots;


// packet transmission time

float packet_time = PACKET_LENGTH * FACTOR / BUS_RATE;


float packet_slots = (float) (int) (packet_time/slot_size) + 1.0;


float infinite = 1.0 * pow (10.0, 30.0);


float rand_size = RAND_MAX;



// array to store delay values

double delay_ci[DEGREES_FR + 1];


// array to store utilization values

double utilization_ci [DEGREES_FR+1];


// array to store throughput values

double throughput_ci [DEGREES_FR+1];
```

```
   // array for collision rate values

   double collision_rate_ci[DEGREES_FR+1];


   double rho        = 0.0;

   double next_event_time;

   double average_delay;

   double collision_rate;

   double throughput;


   int i, j, ic, ii, next_station, next_event, next;




   // T-distribution parameters

   double t_dist_par[] = { 12.706, 4.303, 3.182, 2.776, 2.571,

                            2.447,  2.365, 2.306, 2.262, 2.228 };


   double total_delay;

   double delay_sum, delay_con_int;

   double utilization, utilization_sum, utilization_sqr;

   double utilization_var, utilization_sdv, utilization_con_int;

   double throughput_sum;

   double collision_rate_sum;

   double no_pkts_departed;

   double no_collisions;

//**************************************************************

//  Function:  double traffic(double rho, double arrival_rate)

//  Purpose:   Calculate traffic intensity (rho)
```

```
//*******************************************************
double traffic(double rho, double arrival_rate)
{
   // Calculate the traffic intensity rho and check to see if it
exceeds
   //   the network capacity

   rho = arrival_rate * PACKET_LENGTH * MAX_STATIONS / BUS_RATE;

   if (rho >= 1.0)
   {
      printf("Traffic intensity is too high\n");
      exit(0);
   }

   return rho;
}//end of traffic function




//*******************************************************
//   Function:   void depart(double next_event_time, int next_station,
int next_event)
//  Purpose:   Perform packet departure event.
//*******************************************************
void depart(double next_event_time, int next_station, int next_event)
{
   int id_number = 0;
   double delay = 0.0;
   float time = watch.getTime();

   double collision_end_time = watch.getCollisionEndTime();


   id_number = stations.getQueueId(next_station, 0);

   csmaNetwork.setChannel(0.0);

   stations.decrementQueue(next_station);


   // Push the queue forward


   for (int i = 0; i < stations.queueSize(next_station); i++)

      stations.setQueueId(next_station,                              i,
stations.getQueueId(next_station, i+1));


   stations.setQueueId(next_station,   stations.queueSize(next_station),
0);

   delay = time - ids.getStartTime(id_number);

   total_delay += delay;

   ids.setIdList(id_number, 0);


   no_pkts_departed += 1.0;


   utilization += packet_slots;


   // Schedule the next event for transmission attempt by the station

   events.setEventTime(next_station, 3, infinite);

   if (stations.queueSize(next_station) > 0)

   {

      events.setEventTime(next_station, 1, time + 1.0);

      if (events.getEventTime(next_station, 1) <= collision_end_time)
```

```
         events.setEventTime(next_station,   1,   collision_end_time   +
1.0);
   }
   else
   {
      events.setEventTime(next_station, 1, infinite);
      events.setEventTime(next_station, 2, infinite);
   }
}


//**************************************************************
//   Function:  void transmit(double next_event_time, int next_station,
int next_event)
//  Purpose:   Perform packet transmission
//**************************************************************
void transmit(double next_event_time, int next_station, int next_event)
{
   int no_trans = 0;
   double x = 0.0;
   double collision_end_time = 0.0;
   double time = watch.getTime();
   float rand_size = RAND_MAX;

   // Check to see if a collision has take place.
   for (int i=0; i < MAX_STATIONS; i++)
      if (events.getEventTime(i, 2) == time) no_trans++;

   if (no_trans > 1)
```

```
    {
       watch.setCollisionEndTime(time + JAM_PERIOD + 2.0);
       no_collisions++;

       collision_end_time = watch.getCollisionEndTime();
       for (int i = 0; i < MAX_STATIONS; i++)
       {
          if (events.getEventTime(i, 2) == time)
          {
             events.setEventTime(i, 2, infinite);
             x = (float) rand();
             x = x/rand_size;
             double backoff_time = (double) (int) (x * MAX_BACKOFF);
             if (backoff_time < 1.0) backoff_time = 1.0;

             events.setEventTime(i,      1,      collision_end_time      +
backoff_time);
          }
          if (events.getEventTime(i, 1) <= collision_end_time)
          {
             x = (float) rand();
             x = x/rand_size;
             double backoff_time = (double) (int) (x * MAX_BACKOFF);
             if (backoff_time < 1.0) backoff_time = 1.0;

             events.setEventTime(i,      1,      collision_end_time      +
backoff_time);
          }
       }
    }

    else

    {

       if (csmaNetwork.isChannelFree())

       {

          events.setEventTime(next_station, 3, time + packet_slots);

          events.setEventTime(next_station, 2, infinite);

          csmaNetwork.setChannel(1.0);

       }

       else

       {

          if (!csmaNetwork.isPersistent())

          {

             x = (float) rand();

             x = x/rand_size;

             double backoff_time = (double) (int) (x * MAX_BACKOFF);

             if (backoff_time < 1.0) backoff_time = 1.0;

             events.setEventTime(next_station, 1, time + backoff_time);

             if      (events.getEventTime(next_station,      1)      <=
collision_end_time)

                events.setEventTime(next_station,  1,  collision_end_time
+ backoff_time);


             events.setEventTime(next_station, 2, infinite);

          }

          else

          {

             events.setEventTime(next_station, 1, time + 1.0);
```

```
            if      (events.getEventTime(next_station,      1)      <=
collision_end_time)
               events.setEventTime(next_station, 1, collision_end_time
+ 1.0);
            events.setEventTime(next_station, 2, infinite);
         }
      }
   }

}



//*****************************************************
//   Function:   void attempt(double next_event_time, int next_station,
int next_event)
//  Purpose:   Perform attempt event
//*****************************************************
void attempt(double next_event_time, int next_station, int next_event)
{
   int no_attempts = 0;
   int select_flag = 0;
   double x = 0.0;
   double select_prob = 0.0;
   double time = watch.getTime();
   double collision_end_time = watch.getCollisionEndTime();
   float rand_size = RAND_MAX;

   for (int i = 0; i < MAX_STATIONS; i++)
   {
```

```
   if (events.getEventTime(i, 1) == time)
   {
      no_attempts++;
      stations.setIdAttemptStn(no_attempts-1, i);
   }
}

select_flag = 0;
if (no_attempts > 1)
{
   x = (float) rand();
   x = x/rand_size;
   for (int i=0; i < no_attempts; i++)
   {
      select_prob = (double) (i+1) / ((double) no_attempts);
      if (x <= select_prob)
      {
         next_station = stations.getIdAttemptStn(i);
         select_flag = 1;
      }

      if (select_flag == 1) continue;
   }
}

if (csmaNetwork.isChannelFree())
{
   if (!csmaNetwork.isPersistent())
```

```
      {
         events.setEventTime(next_station, 2, time + 1.0);
         events.setEventTime(next_station, 1, infinite);
      }
      else
      {
         x = (float) rand();
         x = x/rand_size;
         if (x < csmaNetwork.getPersistence())
         {
            events.setEventTime(next_station, 2, time + 1.0);
            events.setEventTime(next_station, 1, infinite);
         }
         else
         {
            events.setEventTime(next_station, 1, time + 1.0);
            if       (events.getEventTime(next_station,       1)       <=
collision_end_time)
               events.setEventTime(next_station,  1,  collision_end_time
+ 1.0);
            events.setEventTime(next_station, 2, infinite);
         }
      }
   }

   if (!csmaNetwork.isChannelFree())
   {
      if (!csmaNetwork.isPersistent())
      {
         x = (float) rand();

         x = x/rand_size;

         double backoff_time = (double) (int) (x * MAX_BACKOFF);

         if (backoff_time < 1.0) backoff_time = 1.0;

         events.setEventTime(next_station, 1, time + backoff_time);

         if        (events.getEventTime(next_station,        1)        <=
collision_end_time)

            events.setEventTime(next_station,  1,  collision_end_time  +
backoff_time);


         events.setEventTime(next_station, 2, infinite);

      }

      else

      {


      events.setEventTime(next_station, 1, time + 1.0);

      if (events.getEventTime(next_station, 1) <= collision_end_time)

         events.setEventTime(next_station,   1,   collision_end_time   +
1.0);

      events.setEventTime(next_station, 2, infinite);

      }

   }

}


//***********************************************************

//   Function:   void arrival(double next_event_time, int next_station,
int next_event)

//  Purpose:   Perform arrival event

//***********************************************************
```

```
void arrival(double next_event_time, int next_station, int next_event)
{
   int id_number = 0;
   double x = 0.0;
   double time = watch.getDTime();
   double d_clock = watch.getDTime();
   double collision_end_time = watch.getCollisionEndTime();
   float rand_size = RAND_MAX;

   // Select an identification for the arriving message
   id_number = ids.SelectIdNumber();

   stations.incrementQueue(next_station);

   stations.setQueueId(next_station,   stations.queueSize(next_station)-
1, id_number);
   ids.setStartTime(id_number, time);

   if (stations.queueSize(next_station) == 1)
   {
      events.setEventTime(next_station, 1, d_clock);

      if (events.getEventTime(next_station, 1) <= collision_end_time)
         events.setEventTime(next_station,   1,   collision_end_time   +
1.0);
   }

   // Schedule the next "arrival" event
   for (;;)
   {
      x = (float) rand();
      if (x != 0.0) break;
   }

   double logx = -log(x/rand_size) * FACTOR / arrival_rate_slots;
   events.setEventTime(next_station, next_event, time + logx);


}

//*****************************************************************
//   Function:   void process(double next_event_time, int next_station,
int next_event)
//  Purpose:   Determine type of event that occurs next.
//*****************************************************************
void process(double next_event_time, int next_station, int next_event)
{
   watch.setTime(next_event_time);
   if ( next_event < 0 || next_event > 3 )
   {
      printf("An event was not recognized.  Check the event-list");
      return;

   }

   while (watch.getDTime() <= watch.getTime())
```

```
      watch.incrementDTime();

   switch (next_event)
   {
      case 0: // This is an arrival event.
      {
         arrival(next_event_time, next_station, next_event);
         break;
      }
      case 1: // This is an attempt event.
      {
         attempt(next_event_time, next_station, next_event);
         break;
      }
      case 2: // This is a transmission event.
      {
         transmit(next_event_time, next_station, next_event);
         break;
      }
      case 3: // This is a departure event.
      {
         depart(next_event_time, next_station, next_event);
         break;
      }
   } // switch
}
```

```
//************************************************************
//  Function:  void compute(double utilization_ci[], double delay_ci[],
//                                       double  throughput_ci[],  double
collision_rate_ci[])
//  Purpose:   Compute statistics
//************************************************************
void compute(double utilization_ci[], double delay_ci[],
                        double          throughput_ci[],          double
collision_rate_ci[])

{
   double delay_sqr, delay_var, delay_sdv;

   delay_sum = 0.0;
   delay_sqr = 0.0;
   utilization_sum = 0.0;
   utilization_sqr = 0.0;
   throughput_sum = 0.0;
   collision_rate_sum = 0.0;

   for (int ic = 0; ic <= DEGREES_FR; ic++)
   {
      delay_sum += delay_ci[ic];
      delay_sqr += pow(delay_ci[ic],2.0);
      utilization_sum += utilization_ci[ic];
      utilization_sqr += pow(utilization_ci[ic],2.0);
      throughput_sum += throughput_ci[ic];
      collision_rate_sum += collision_rate_ci[ic];
   }

   delay_sum = delay_sum / (DEGREES_FR + 1);
   delay_sqr = delay_sqr / (DEGREES_FR + 1);
   delay_var = delay_sqr - pow(delay_sum,2.0);
   delay_sdv = sqrt(delay_var);
   delay_con_int      =      delay_sdv      *      t_dist_par[DEGREES_FR-
1]/sqrt(DEGREES_FR);
   utilization_sum = utilization_sum / (DEGREES_FR + 1);
   utilization_sqr = utilization_sqr / (DEGREES_FR + 1);
   utilization_var = utilization_sqr - pow(utilization_sum,2.0);
   utilization_sdv = sqrt(utilization_var);
   utilization_con_int   =   utilization_sdv   *   t_dist_par[DEGREES_FR-
1]/sqrt(DEGREES_FR);
   throughput_sum = throughput_sum / (DEGREES_FR + 1);
   collision_rate_sum = collision_rate_sum / (DEGREES_FR + 1);
}

//***************************************************************
//  Function:  void output(double rho, double arrival_rate)
//  Purpose:   Output data
//***************************************************************
void output(double rho, double arrival_rate)
{
   printf("For an arrival rate = %g\n", arrival_rate);
   printf("The traffic intensity = %g\n" , rho);
   printf("The average delay = %g" , delay_sum);
   printf(" +- %g\n" , delay_con_int);
   printf("The utilization = %g" , utilization_sum);
   printf(" +- %g\n" , utilization_con_int);

   printf("The throughput = %g\n" , throughput_sum);

   printf("The collision rate = %g\n" , collision_rate_sum);

   printf("\n");

}


//**********************************************************

//  Function:  void initialize()

//  Purpose:   Initialize class objects.

//**********************************************************

void initialize()

{

   csmaNetwork.setChannel(0.0);

   watch.reset();

   events.initialize();

   stations.initialize();

   ids.initialize();

}





int main() {

   float rand_size = RAND_MAX;


  // for ( ii = 0; ii < 30; ii++) {}

   for (ii = 0; ii < 10; ii++) {


      arrival_rate = arrival_rate + 20.0;
```

```
    for (ic = 0; ic <= DEGREES_FR; ic++)
    {

       // Initialize all variables to their appropriate values.

          rho                  = 0.0;
       utilization          = 0.0;
       no_pkts_departed     = 0.0;
       total_delay          = 0.0;
       next_event_time      = 0.0;
       average_delay        = 0.0;
       no_collisions        = 0;

       initialize();

       arrival_rate_slots = arrival_rate * slot_size;



             // Compute traffic intensity
       rho = traffic(rho, arrival_rate);

       while (no_pkts_departed < MAX_PACKETS)
       {
               // Scan the event list and pick the next event to be
executed.
          next_event_time = events.nextEventTime();
          next_station    = events.nextStation();
```

```
            next_event      = events.nextEvent();

            process(next_event_time, next_station, next_event);

         }

               // A simulation run is to be terminated


         utilization = utilization / watch.getTime();

         average_delay = total_delay * slot_size / (no_pkts_departed *
FACTOR);

         throughput = no_pkts_departed * FACTOR / (watch.getTime() *
slot_size);

         collision_rate = (double) no_collisions * FACTOR /
(watch.getTime() * slot_size);

         utilization_ci[ic] = utilization;

         delay_ci[ic] = average_delay;

         throughput_ci[ic] = throughput;

         collision_rate_ci[ic] = collision_rate;

      }


      compute(utilization_ci, delay_ci, throughput_ci,
collision_rate_ci);

      output(rho, arrival_rate);


   }


   return 0;

}
```

C-2 CONSTANTS.H

```
#ifndef _CONSTANTS_H

#define _CONSTANTS_H


# include <math.h>


# define MAX_STATIONS        10    // Number of stations

# define BUS_RATE       2000000.0  // Transmission rate in bps

# define PACKET_LENGTH     1000.0  // Packet length (bits)

# define BUS_LENGTH        2000.0  // Bus length in meters

# define MAX_BACKOFF         15.0  // Backoff period in slots

# define PERSIST              0.0  // Persistence

# define JAM_PERIOD           5.0  // Jamming period

# define MAX_PACKETS       100      // Maximum packets to be transmitted
in a simulation run

# define FACTOR            1000.0  // A factor used for changing units
of time

# define MAX_Q_SIZE        500    // Maximum queue size

# define ID_SIZE          50      // Size of the identity array

# define DEGREES_FR           5    // Degrees of freedom

# define SIMULATION_RUNS = 10;



#endif
```

C-3 CLOCK.CPP

```
#include <math.h>

#include <stdio.h>

#include <stdlib.h>


#include "clock.h"


Clock :: Clock() {

   setTime(0.0);

   setDTime(0.0);

   setCollisionEndTime(0.0);

}


Clock :: Clock(float new_time, float newDTime, float newCollEndTime) {

   setTime(new_time);

   setDTime(newDTime);

   setCollisionEndTime(newCollEndTime);

}


float Clock :: getTime() {

   return clock;

}


float Clock :: getDTime() {

   return d_clock;

}


float Clock :: getCollisionEndTime() {
   return collision_end_time;
}

void Clock :: setTime(float new_time) {
   if (new_time < 0.0) {
      printf("Invalid time. \n");
      exit(1);
   }

   clock = new_time;
}

void Clock :: setDTime(float new_time){
   if (new_time < 0.0) {
      printf("Invalid time. \n");
      exit(1);
   }

   d_clock = new_time;
}

void Clock :: setCollisionEndTime(float new_time){
   if (new_time < 0.0) {
      printf("Invalid time. \n");
      exit(1);
   }

   collision_end_time = new_time;
```

```
}


void Clock :: incrementDTime() {
   d_clock++;
}


void Clock :: reset() {
   setTime(0.0);
   setDTime(0.0);
   setCollisionEndTime(0.0);
}
```

C-4 CLOCK.H

```
#ifndef _CLOCK_H
#define _CLOCK_H

#include <math.h>



class Clock {


public:
   Clock :: Clock();
   Clock :: Clock(float time, float dtime, float cetime);
   float Clock :: getTime();
   float Clock :: getDTime();
   float Clock :: getCollisionEndTime();
   void Clock :: setTime(float new_time);
   void Clock :: setDTime(float new_time);
   void Clock :: setCollisionEndTime(float new_time);
   void Clock :: incrementDTime();
   void Clock :: reset();

   float clock;  //simulation clock
   float d_clock;  //clock to determine slot timing
   float collision_end_time;  //the time at which collision ends

};

#endif
```

C-5 EVENTLIST.CPP

```
#include <math.h>

#include <stdio.h>

#include <stdlib.h>


#include "eventList.h"


float infiniteB = 1.0 * pow (10.0, 30.0);


EventList :: EventList() {

   //for all stations...

         for (int q = 0; q < MAX_STATIONS; q++)

         {

            for (int r = 0; r < 4; r++)

            {

               //set event times to infinite

               event_time[q][r] = infiniteB;

               //get a random number

               x = (float) rand();

               x = x * FACTOR/rand_size;

               //set first event time to random value

               if (r == 0) event_time[q][r] = x;

            }//end of r for loop


         }//end of q loop


}
```

```
float EventList :: getEventTime(int next_station, int eventType) {
   return event_time[next_station][eventType];
}


float  EventList  ::  nextEventTime()  {//set  next  event  time  infinitely
large
   next_event_time = infiniteB;

   //scan all stations'...
   for (int i = 0; i < MAX_STATIONS; i++)
   {
      //queues
      for (int j = 0; j < 4; j++)
      {
         //if next event time is after station's event's time
         if (next_event_time > event_time[i][j])
         {
            //set next event time to station's event's time
            next_event_time = event_time[i][j];

            //set station and event of next event to perform
            next_station = i;
            next_event   = j;
         }//end of if

      }//end of j for

   }//end of i for
   return next_event_time;
}



int  EventList :: nextStation() {
   next_event_time = infiniteB;

   //scan all stations'...
   for (int i = 0; i < MAX_STATIONS; i++)
   {
      //queues
      for (int j = 0; j < 4; j++)
      {
         //if next event time is after station's event's time
         if (next_event_time > event_time[i][j])
         {
            //set next event time to station's event's time
            next_event_time = event_time[i][j];

            //set station and event of next event to perform
            next_station = i;
            //next_event   = j;
         }//end of if

      }//end of j for

   }//end of i for
   return next_station;
}



int  EventList :: nextEvent() {
   next_event_time = infiniteB;

   //scan all stations'...
   for (int i = 0; i < MAX_STATIONS; i++)
   {
      //queues
      for (int j = 0; j < 4; j++)
      {
         //if next event time is after station's event's time
         if (next_event_time > event_time[i][j])
         {
            //set next event time to station's event's time
            next_event_time = event_time[i][j];

            //set station and event of next event to perform
            //next_station = i;
            next_event   = j;
         }//end of if

      }//end of j for

   }//end of i for
   return next_event;
}


void EventList :: setEventTime(int next_station, int eventType, float
clock) {
   event_time[next_station][eventType] = clock;
}


void EventList :: initialize() {
   double x = 0.0;
   // Initialize the event list
   for (int i = 0; i < MAX_STATIONS; i++)
   {
      for (int j = 0; j < 4; j++)
      {
         event_time[i][j] = infiniteB;
         x = (float) rand();
         if (j == 0)
            event_time[i][j] = x;
      }
   }
}
```

C-6 EVENTLIST.H

```
#ifndef _EVENTLIST_H

#define _EVENTLIST_H


#include <math.h>

#include "constants.h"



class EventList {


public:

   EventList :: EventList();

   float EventList :: getEventTime(int next_station, int eventType);

   float EventList :: nextEventTime();

   int  EventList :: nextStation();

   int  EventList :: nextEvent();

   void  EventList  ::  setEventTime(int  next_station,  int  eventType,
float clock);

   void EventList :: initialize();

   float event_time [MAX_STATIONS] [4];  //time of occurrence of event

   float x;

   float rand_size;

   float next_event_time;

   int next_station;

   int next_event;

};


#endif
```

C-7 IDLIST.CPP

```
#include <math.h>

#include <stdio.h>

#include <stdlib.h>



#include "idList.h"


IdList :: SelectIdNumber() {

  // Select an identification for the arriving message


   //set identification number to -1

   id_number = -1;


   //for all i less than number of ids

   for (int i = 0; i < ID_SIZE; i++)

   {

      //if 'flag' at i is zero

      if (id_list[i] == 0)

      {

         //set id number of packet to i

         id_number = i;

         //set 'flag' at i to one

         id_list[i] = 1;

         break;

      }


      //if id number was available continue
      if (id_number != -1) continue;
   }

   //if id number wasn't changed, either id's are gone or there
   //is a problem with the id list...
   if (id_number == -1)
   {
      //inform user to check id list and stop program
      printf("Check the ID-list.");
      exit(1);
   }

   return id_number;
}

IdList :: IdList() {
    //set start time and id list of all stations to zero
         for (int k = 0; k < ID_SIZE; k++)
         {
            start_time[k] = 0.0;
            id_list[k] = 0;
         }
}


void IdList :: setStartTime(int id_number, float clock){
   start_time [id_number] = clock;
}
```

```
float IdList :: getStartTime(int id_number) {

   return start_time[id_number];

}



IdList :: setIdList(int id_number, int number) {

   id_list[id_number] = number;

}


void IdList :: initialize(){

    for (int i = 0; i < ID_SIZE; i++)

    {

       id_list[i] = 0;

       start_time[i] = 0.0;  // Starting time of these packets

    }

}
```

C-8 IDLIST.H

```
#ifndef _IDLIST_H

#define _IDLIST_H


#include <math.h>

#include "constants.h"



class IdList {


public:

   IdList :: IdList();

   int IdList :: SelectIdNumber();

   IdList :: setIdList(int id_number, int number);

   void IdList :: setStartTime(int id_number, float clock);

   float IdList :: getStartTime(int id_number);

   float start_time [ID_SIZE];     // starting time of packet

   int id_list [ID_SIZE];          // array of id_numbers

   int id_number;

   void IdList :: initialize();


};


#endif
```

C-9 NETWORK.CPP

```
#include <math.h>

#include <stdio.h>

#include <stdlib.h>


#include "network.h"


Network :: Network(float persistence) {

   if (persistence >= 0.0 && persistence <= 1.0)

      p = persistence;


}


void Network :: setPersistence(float persistence)

   {

      p = persistence;

   }



bool Network :: isPersistent()

   {

      if (p == 0.0)

         return false;

      else

         return true;

   }
```

```
float Network :: getPersistence()
   {
      return p;
   }



void Network :: setChannel(float status)
   {
      ch_busy = status;
   }



bool Network :: isChannelFree()
   {
      if (ch_busy == 0.0)
         return true;
      else
         return false;
   }
```

C-10 NETWORK.H

```
#ifndef _NETWORK_H

#define _NETWORK_H


#include <math.h>

#include "constants.h"



class Network {


public:

   Network :: Network(float persistence);

   void Network :: setPersistence(float persistence);

   bool Network :: isPersistent();

   void Network :: setChannel(float status);

   bool Network :: isChannelFree();

   float Network :: getPersistence();


   float start_time [ID_SIZE];   // starting time of packet

   int id_list [ID_SIZE];        // array of id_numbers

   int id_number;

   float p;

   float ch_busy;



};


#endif
```

C-11 STATIONLIST.CPP

```
#include <math.h>

#include <stdio.h>

#include <stdlib.h>



#include "stationList.h"


StationList :: StationList() {

   //set queue size at all stations to zero

   for (int i = 0; i < MAX_STATIONS; i++)

      queue_size[i]=0;

}



StationList :: ~StationList() {

}


StationList :: incrementQueue(int next_station) {

   //increment station's queue size (to hold new packet)

   queue_size[next_station] ++ ;


   //if station's queue size is too big...

   if (queue_size[next_station] > MAX_Q_SIZE)

   {

      //inform user and stop program

      printf("The    queue    size    is    large    and    is    =    %d\n",
queue_size[next_station]);
```

```
      exit(1);
   }
}

StationList :: decrementQueue(int next_station) {
   queue_size[next_station]--;
}

StationList :: initialize(){
    // Initialize queue sizes from all stations to 0
    for (int i = 0; i < MAX_STATIONS; i++)
    {
       queue_size[i] = 0;
    }
}



int StationList :: queueSize(int next_station) {

   return queue_size [next_station];
}

void StationList :: setQueueId(int next_station, int size, int id) {
   queue_id[next_station][size] = id;
}

int  StationList :: getQueueId(int next_station, int size) {
   return queue_id[next_station][size];
```

```
}


void StationList :: setIdAttemptStn(int no_attempts, int next_station)
{
   id_attempt_stn[no_attempts] =  next_station;
}


int StationList ::  getIdAttemptStn(int next_station) {
   return id_attempt_stn[next_station];
}
```

C-12 STATIONLIST.H

```
#include "log_inst.h"


#ifndef _STATIONLIST_H

#define _STATIONLIST_H


#include <math.h>

#include "constants.h"



class StationList {


public:

   StationList :: StationList();

   StationList :: StationList(int max_stats, int max_queue);

   StationList :: incrementQueue(int next_station);

   StationList :: decrementQueue(int next_station);

   int StationList :: queueSize(int next_station);

   void StationList :: setQueueId(int next_station, int size, int id);

   int  StationList :: getQueueId(int next_station, int size);

   void    StationList    ::    setIdAttemptStn(int    no_attempts,    int
next_station);

   int StationList :: getIdAttemptStn(int next_station);

   StationList :: ~StationList();

   StationList :: initialize();


   int queue_size [MAX_STATIONS];            // current queue size at a
station
```

```
   int queue_id [MAX_STATIONS][MAX_Q_SIZE]; // array for id_numbers of
packets
   int  id_attempt_stn  [MAX_STATIONS];            //  array  to  identify
attempting stations
   int next_station;
   int next_event;
   float event_time [MAX_STATIONS] [4];  //time of occurrence of event
   float x;
   float rand_size;
   float next_event_time;
};

#endif
```

APPENDIX D. CSMA/CD UML DIAGRAM

This appendix contains the UML diagrams created by Neil Acantilado for the 2000 winter quarter offering of SW4540 – Software Testing at the Naval Postgraduate School that the authors were simultaneously taking from Prof. J. Bret Michael.

D-1 CONCEPTUAL MODEL

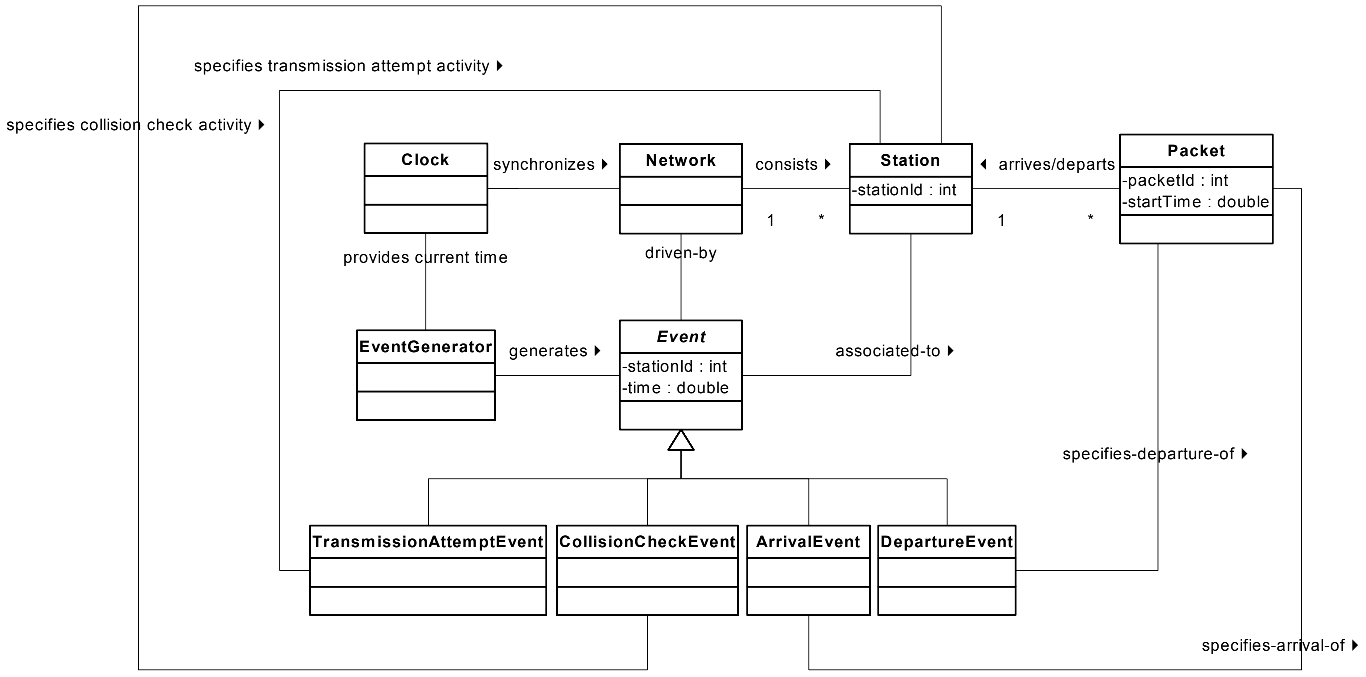


Figure D-1: Conceptual Model

D-2 ACTIVITY DIAGRAM FOR NETWORK SIMULATION

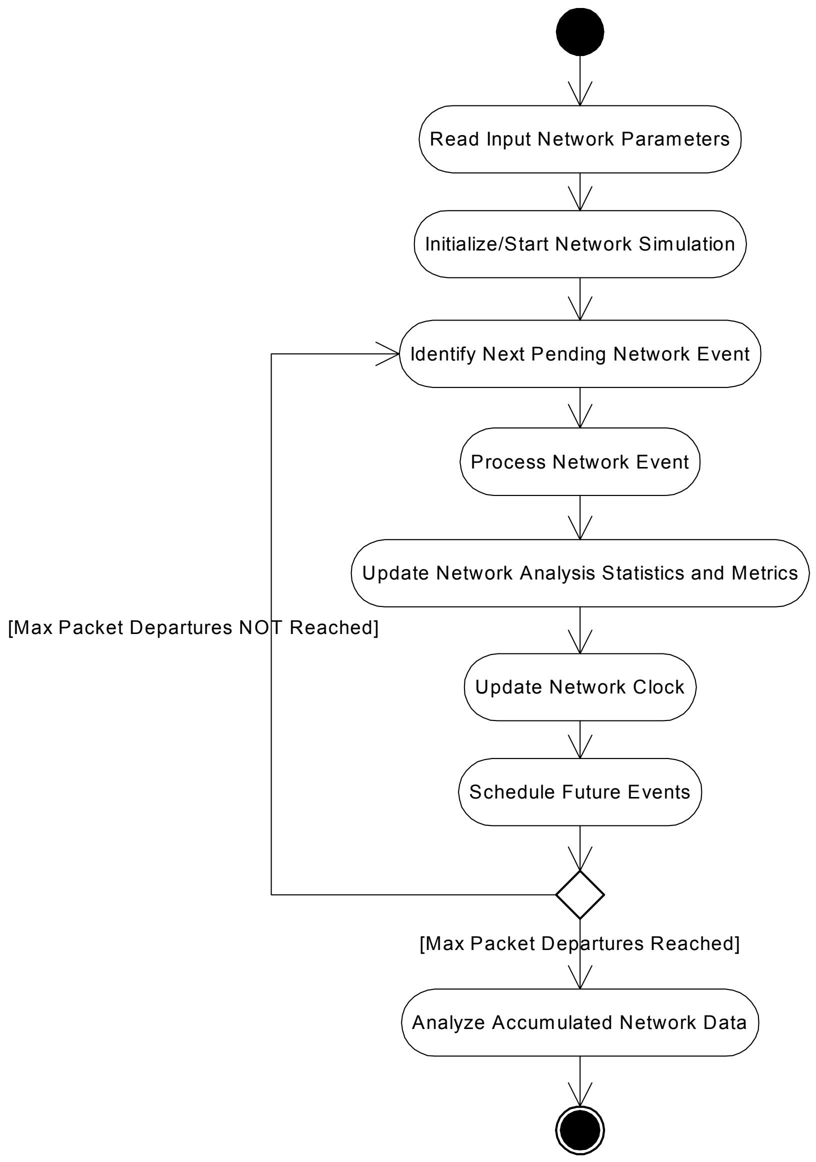

Read Input Network Parameters
Initialize/Start Network Simulation
Identify Next Pending Network Event
Process Network Event
Update Network Analysis Statistics and Metrics
[Max Packet Departures NOT Reached]
Update Network Clock
Schedule Future Events
[Max Packet Departures Reached]
Analyze Accumulated Network Data

Figure D-2: Activity Diagram for Network Simulation

D-3 ACTIVITY DIAGRAM FOR PROCESSING PACKET-ARRIVAL EVENT

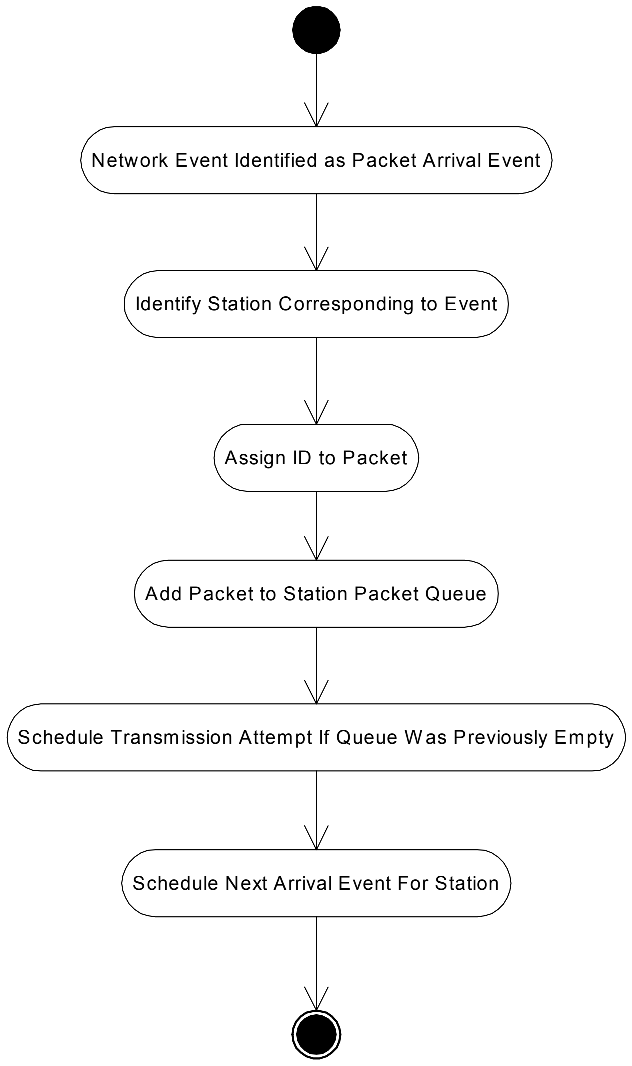

Network Event Identified as Packet Arrival Event
Identify Station Corresponding to Event
Assign ID to Packet
Add Packet to Station Packet Queue
Schedule Transmission Attempt If Queue Was Previously Empty
Schedule Next Arrival Event For Station

Figure D-3: Activity Diagram for Processing Packet-Arrival Events

D-4 ACTIVITY DIAGRAM FOR PROCESSING TRANSMISSION-ATTEMPT EVENTS

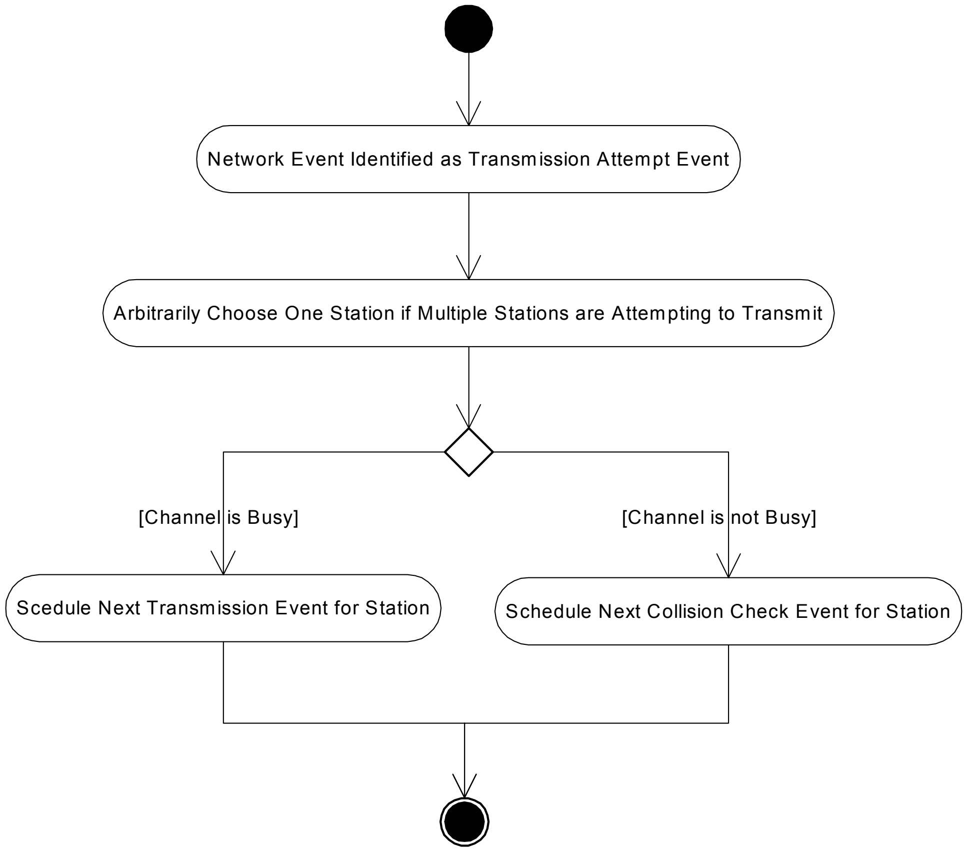

Network Event Identified as Transmission Attempt Event
Arbitrarily Choose One Station if Multiple Stations are Attempting to Transmit
[Channel is Busy]
[Channel is not Busy]
Scedule Next Transmission Event for Station
Schedule Next Collision Check Event for Station

Figure D-4: Activity Diagram for Processing Transmission-Attempt Events

D-5 ACTIVITY DIAGRAM FOR PROCESSING COLLISION-CHECK EVENTS

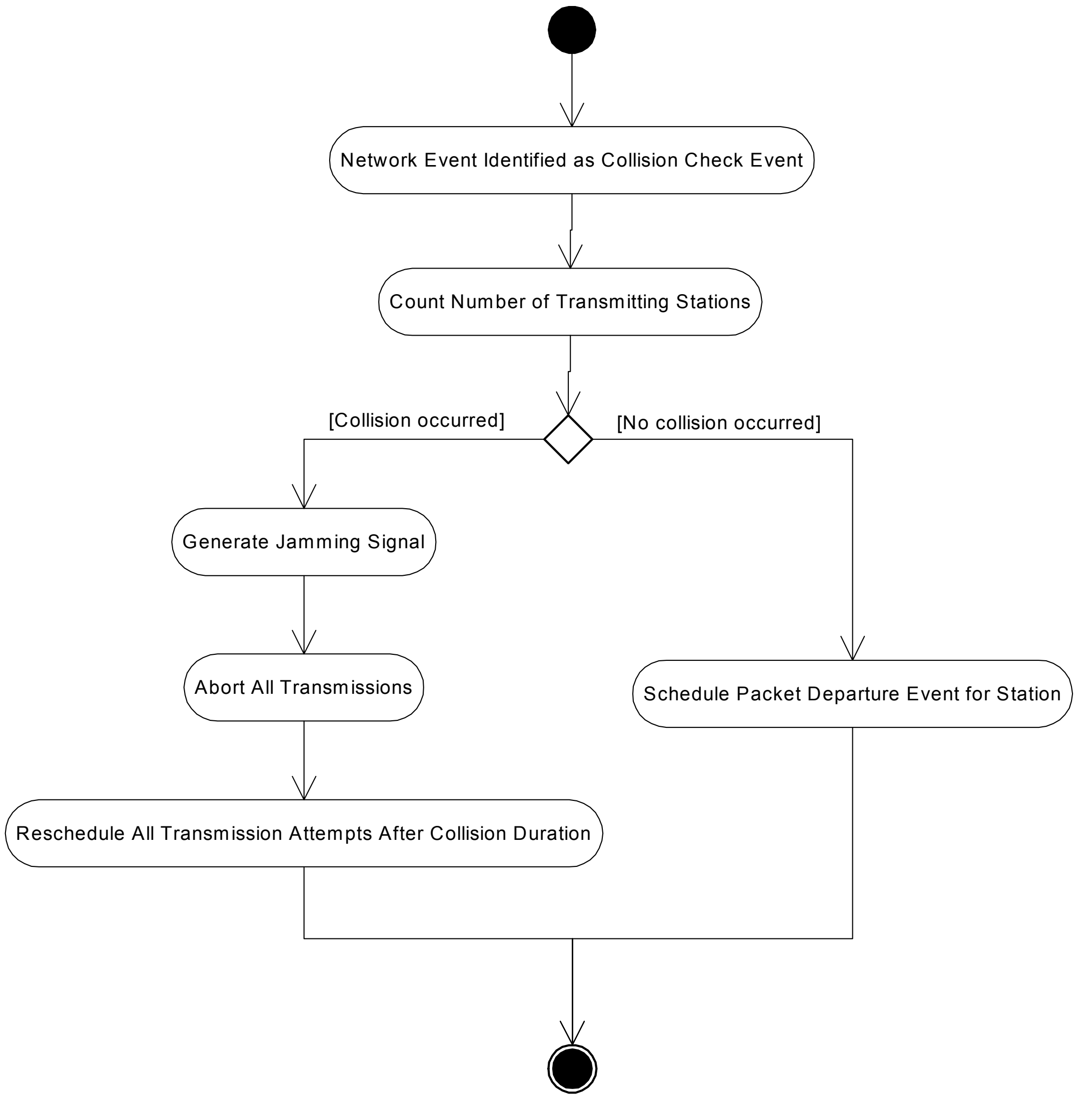

Network Event Identified as Collision Check Event
Count Number of Transmitting Stations
[Collision occurred]
[No collision occurred]
Generate Jamming Signal
Abort All Transmissions
Reschedule All Transmission Attempts After Collision Duration
Schedule Packet Departure Event for Station

Figure D-5: Activity Diagram for Processing Collision-Check Events

D-6 ACTIVITY DIAGRAM FOR PROCESSING PACKET-DEPARTURE EVENTS

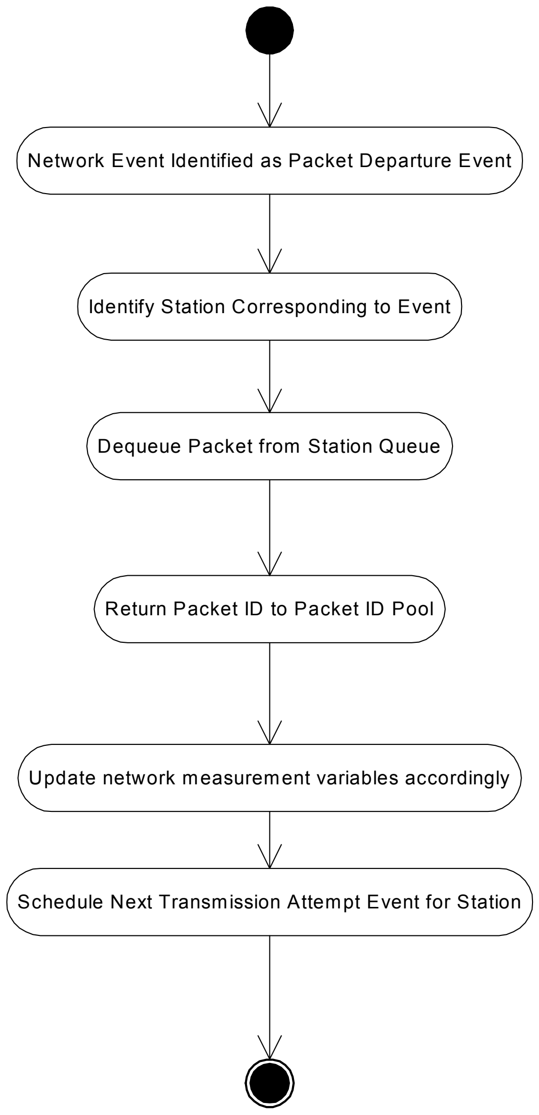

Network Event Identified as Packet Departure Event
Identify Station Corresponding to Event
Dequeue Packet from Station Queue
Return Packet ID to Packet ID Pool
Update network measurement variables accordingly
Schedule Next Transmission Attempt Event for Station

Figure D-6: Activity Diagram for Processing Packet-Departure Events

D-7 DESIGN CLASS DIAGRAM

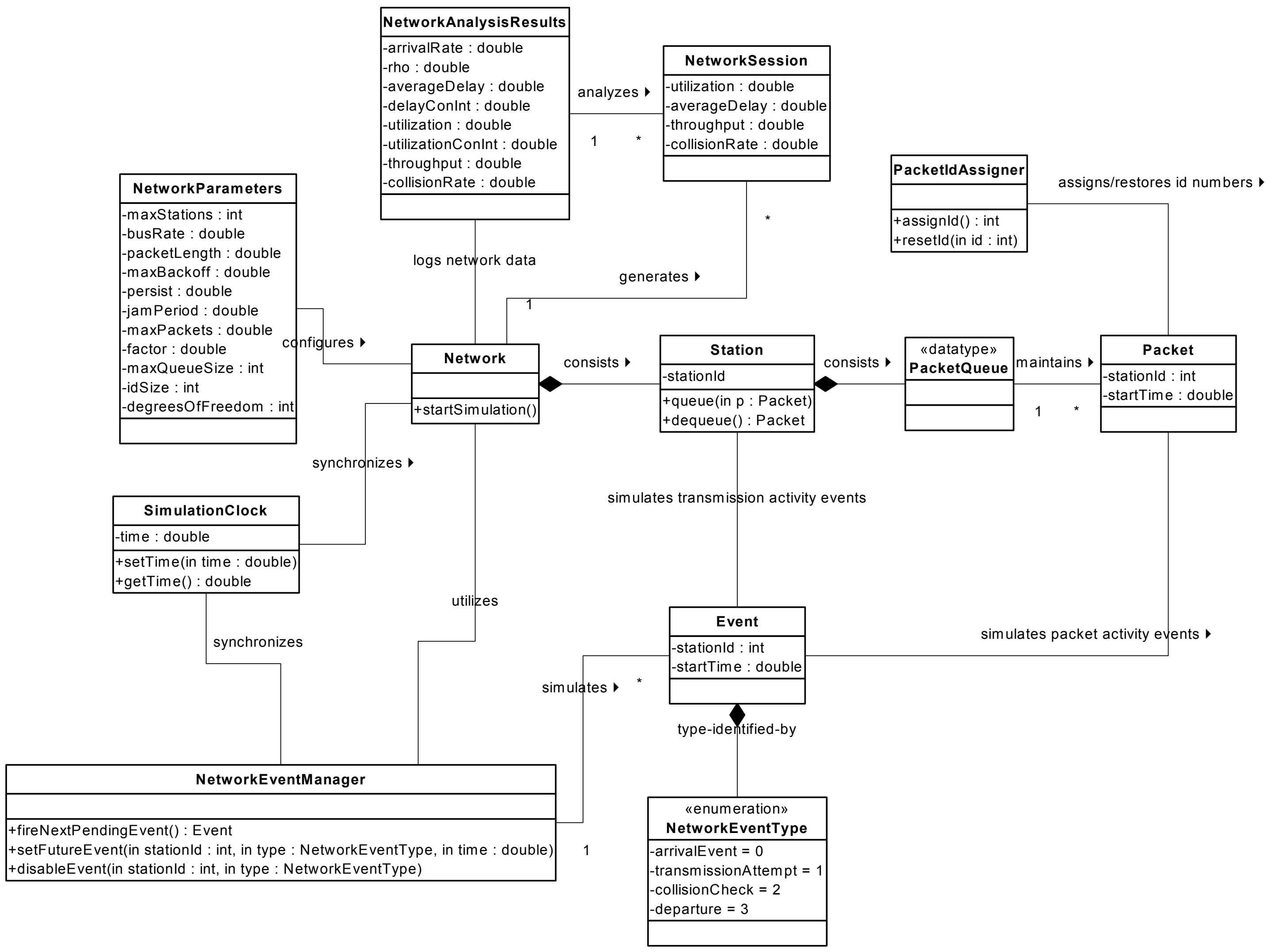


Figure D-7: Design Class Diagram

D-8 INTERACTION DIAGRAM FOR NETWORK SIMULATION

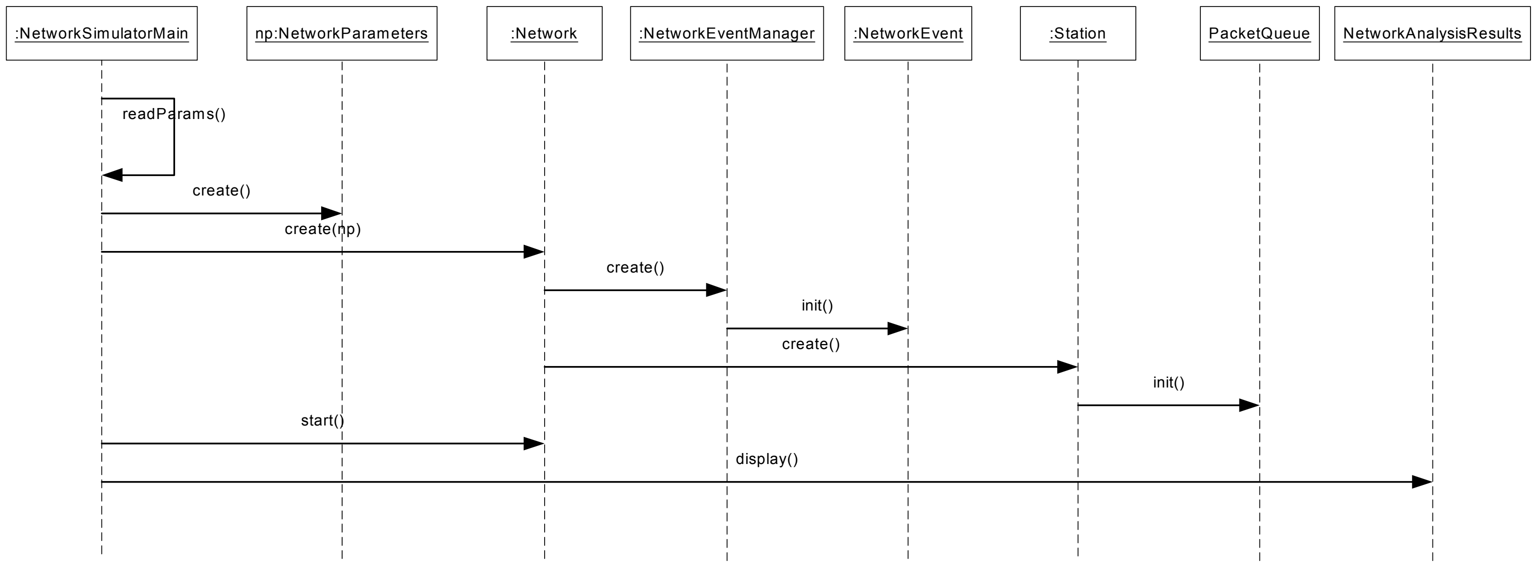


Figure D-8: Interaction Diagram for Network Simulation

D-9 INTERACTION DIAGRAM FOR PROCESSING PACKET-ARRIVAL EVENTS

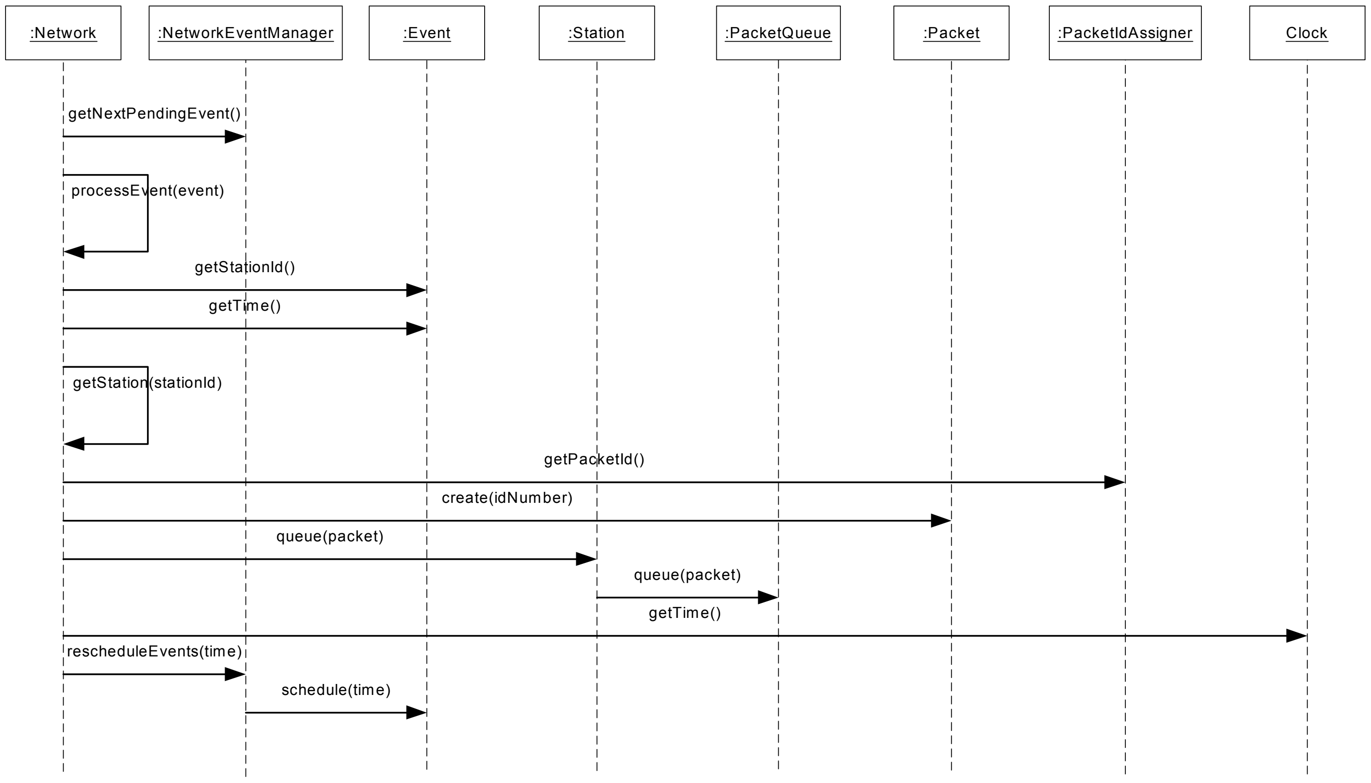


Figure D-9: Interaction Diagram for Processing Packet-Arrival Events

D-10 INTERACTION DIAGRAM FOR PACKET-DEPARTURE EVENT

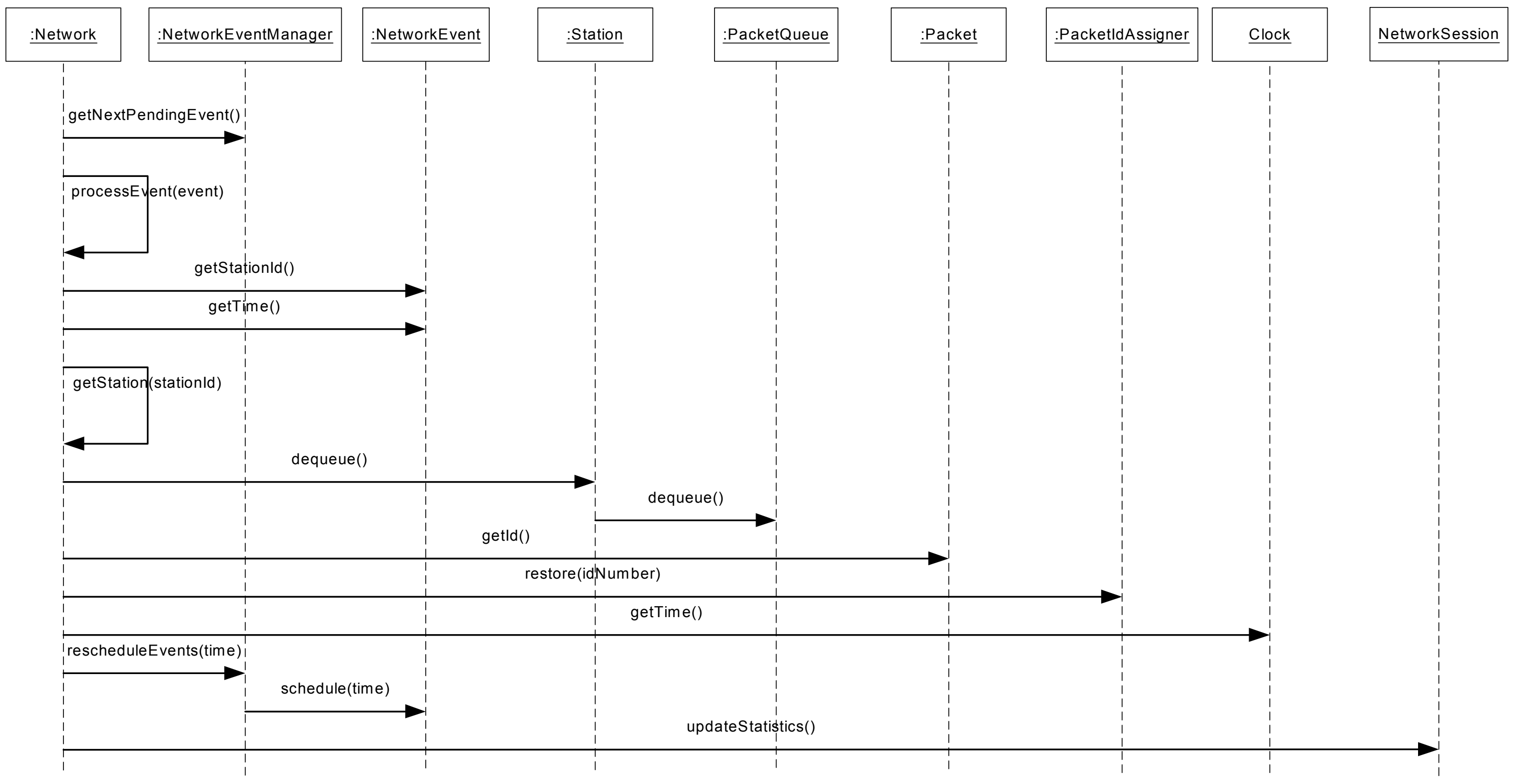


Figure D-10: Interaction Diagram for Packet Departure Event

D-11 GENERIC INTERACTION DIAGRAM FOR PROCESSING TRANSMISSION-ATTEMPT AND COLLISION-ATTEMPT EVENT

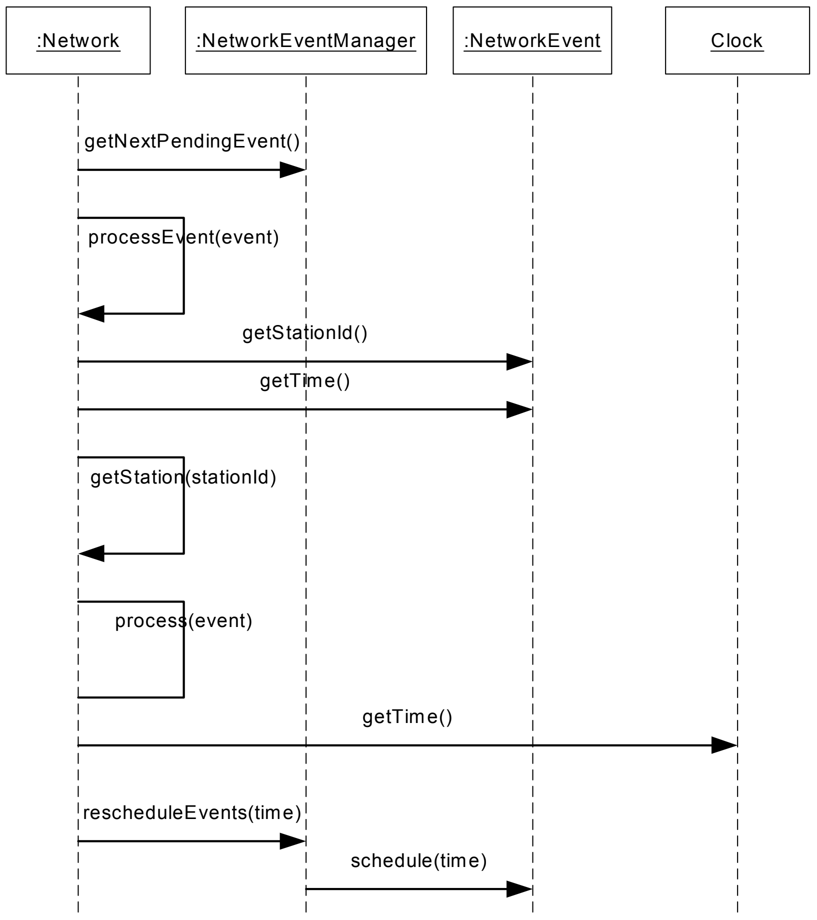


Figure D-11: Generic Interaction Diagram for Processing Transmission-Attempt and Collision Attempt Event

D-12 COLLABORATION DIAGRAM FOR PROCESSING PACKET-ARRIVAL EVENTS

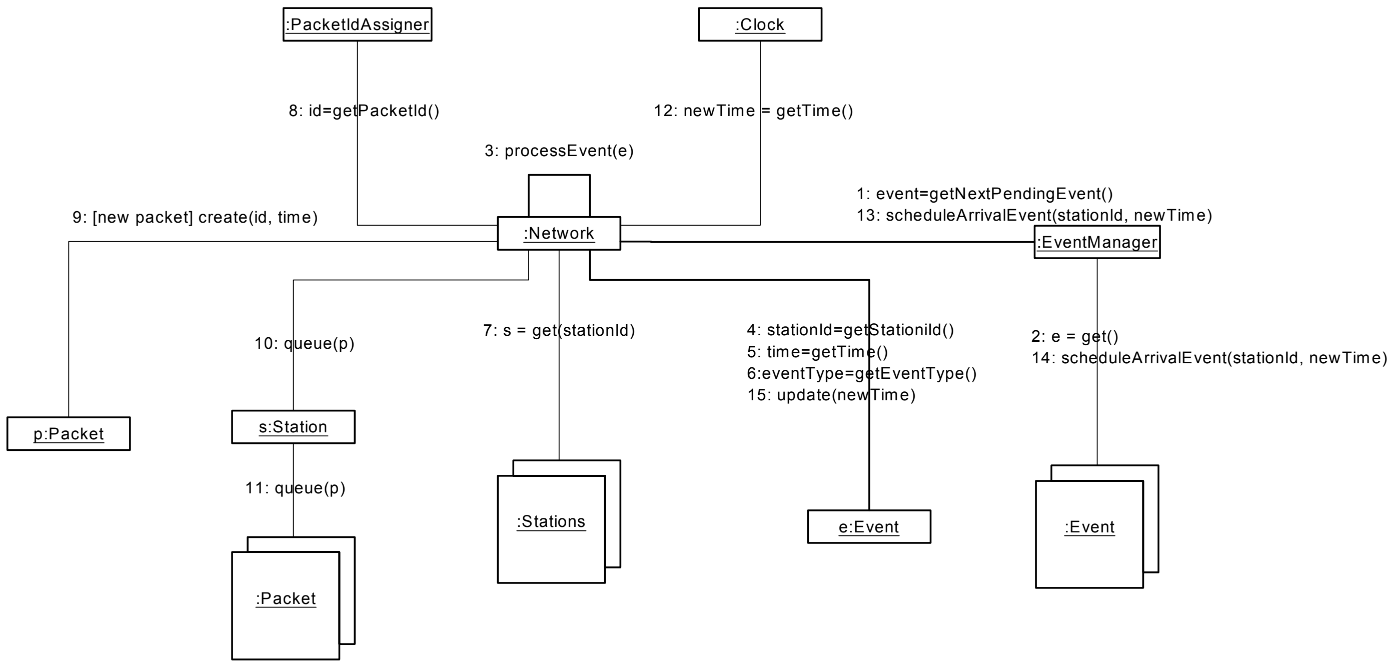


Figure D-12: Collaboration Diagram for Processing Packet-Arrival Events

D-13 COLLABOARION DIAGRAM FOR PROCESSINGG PACKET-DEPARTURE EVENTS

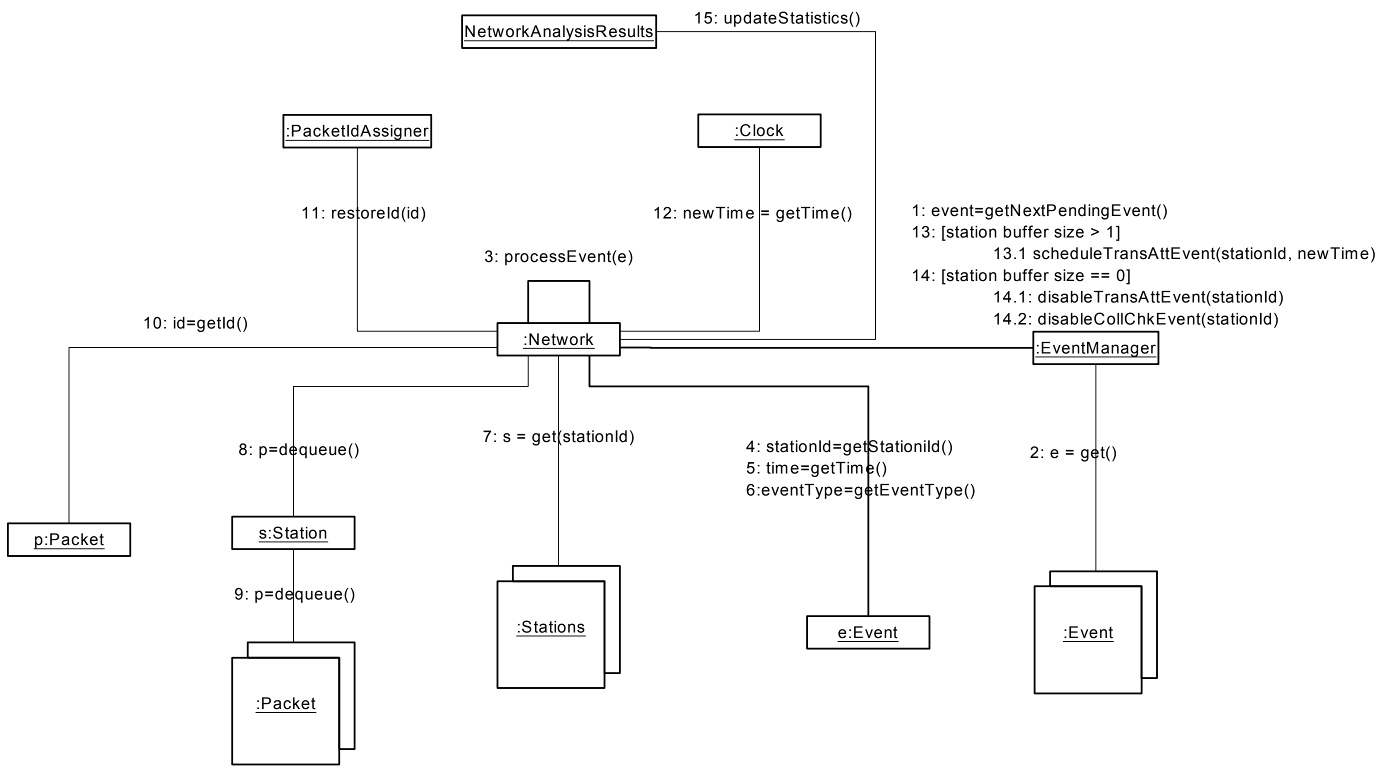


Figure D-13: Collaboration Diagram for Processing Packet-Diagram Events

D-14 COLLABORATION DIAGRAM FOR PROCESSING TRANSMISSION-ATTEMPT EVENTS

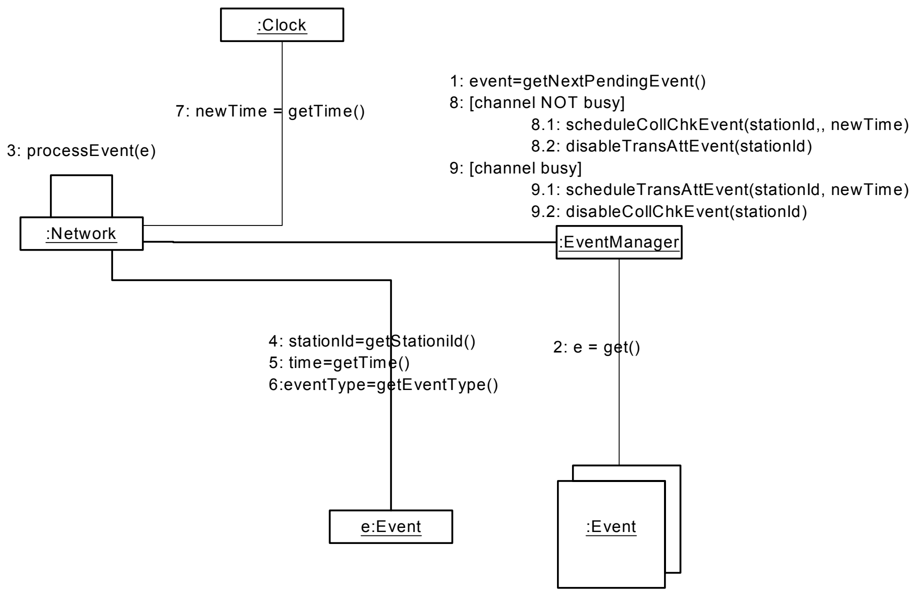

:Clock
7: newTime = getTime()
3: processEvent(e)
:Network
1: event=getNextPendingEvent()
8: [channel NOT busy]
8.1: scheduleCollChkEvent(stationId,, newTime)
8.2: disableTransAttEvent(stationId)
9: [channel busy]
9.1: scheduleTransAttEvent(stationId, newTime)
9.2: disableCollChkEvent(stationId)
:EventManager
4: stationId=getStationiId()
5: time=getTime()
6:eventType=getEventType()
2: e = get()
e:Event
:Event

Figure D-14: Collaboration Diagram for Processing Transmission-Attempt Events

D-15 COLLABORATION DIAGRAM FOR PROCESSING COLLISION-CHECK EVENTS

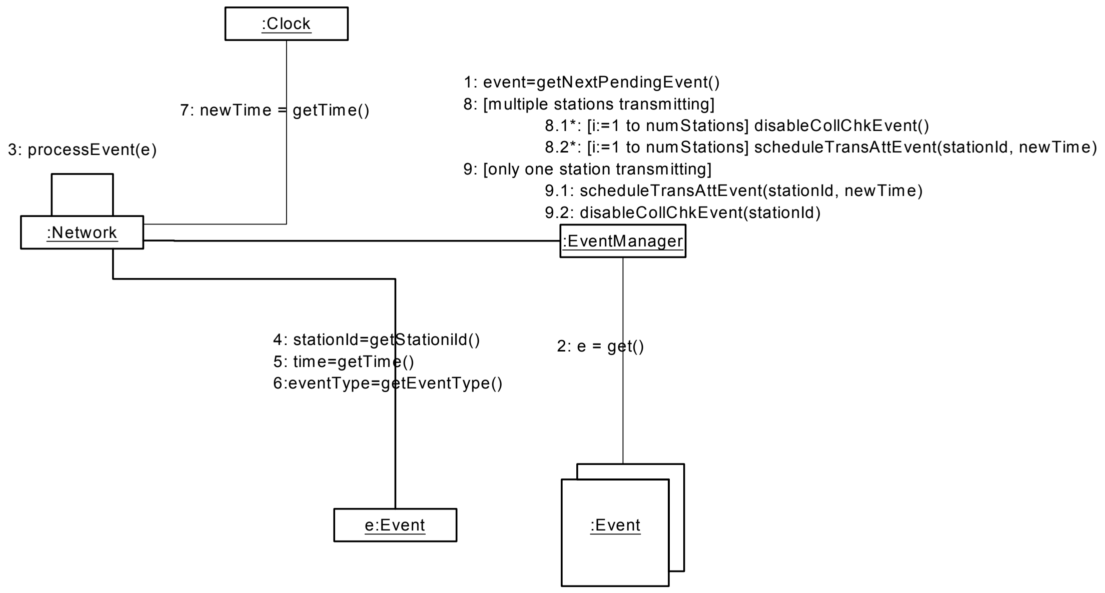

:Clock
7: newTime = getTime()
3: processEvent(e)
:Network
1: event=getNextPendingEvent()
8: [multiple stations transmitting]
8.1*: [i:=1 to numStations] disableCollChkEvent()
8.2*: [i:=1 to numStations] scheduleTransAttEvent(stationId, newTime)
9: [only one station transmitting]
9.1: scheduleTransAttEvent(stationId, newTime)
9.2: disableCollChkEvent(stationId)
:EventManager
4: stationId=getStationiId()
5: time=getTime()
6:eventType=getEventType()
2: e = get()
e:Event
:Event

Figure D-15: Collaboration Diagram for Processing Collision-Check Events

APPENDIX E. TESTING TOOL METRICS RECORD SHEET

This appendix contains a form for recording the measurements obtained when applying our proposed metrics to a software testing tool. If a metric has sub-metrics, blanks are available for recording the sub-metrics directly below the metric.

Testing Tool Metric Record Sheet for ______________ (TESTING TOOL)

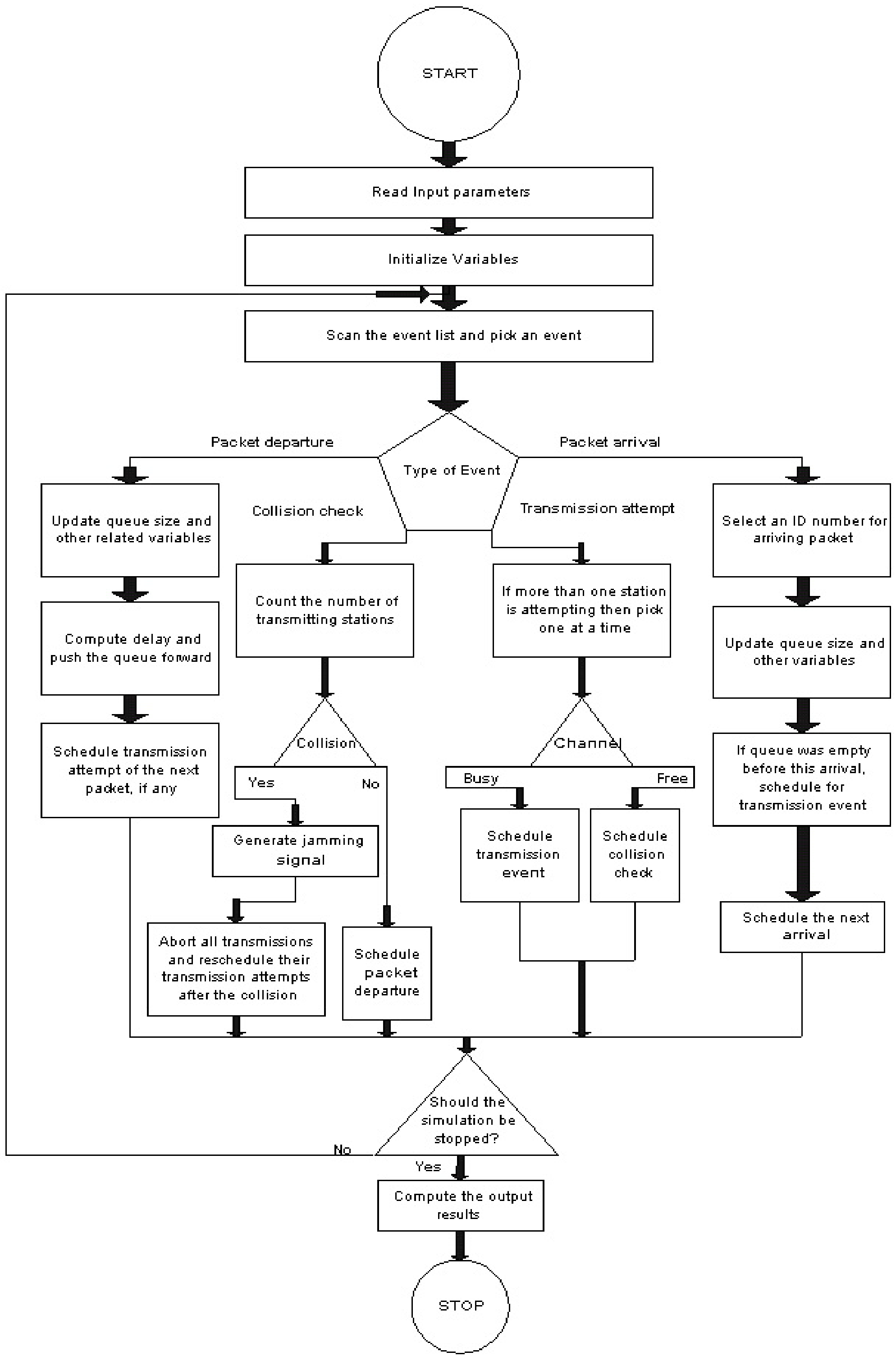
Human Interface Design (HID)		__________
Average Keyboard-to-Mouse Switches (KMS)	__________	
Average Input Fields per Function (IFPF)	__________	
Average Length of Input Fields (ALIF)	__________	
Button Recognition (BR)	__________	
Maturity & Customer Base (MCB)		__________
Maturity (years of tool existence) (M)	__________	
Customer Base (number of users) (CB)	__________	
Projects of similar size using tool (P)	__________	
Tool Management (TM)		__________
Number of Access Levels (NAL)	__________	
Information control methods (ICM)	__________	
Ease of Use (EU)		__________
Learning Time for First-time Users (LTFU)	__________	
Retainability of procedural knowledge by frequent users (RFU)	__________	
Retainability of procedural knowledge by casual users (RCU)	__________	
Operational Time for Frequent Users (OTFU)	__________	
Operational Time for Casual Users (OTCU)	__________	
User Control (UC)		__________
Sum of different portions & portion combinations that can be tested		
Test Case Generation (TCG)		__________
Level of Automated Test Case Generation (ATG)	__________	
Level of Test Case Reuse Functionality (TRF)	__________	
Tool Support (TS)		__________
Average Response Time during normal working hours (ART)	__________	
Average Response Time after hours (ARTAH)	__________	
Average Time to Search Documentation (ATSD)	__________	
Documentation Inadequacy (# of unsuccessful searches) (DI)	__________	
Response to Product Surveys (RPS)	__________	
Reliability (Rel)		__________
Mean Time between Failures (MTF)		
Maximum Number of Parameters (MNP)		__________
Maximum number of parameters allowed in one project		
Response Time (RT)		__________
Average Response Time		

Estimated Return on Investment (EROI)		__________
Estimated Productivity Gain (EPG)	__________	
Estimated Testing Time without tool (ETT)	__________	
Average Cost of One Testing Hour (ACTH)	__________	
Estimated Income Increase (EII)	__________	
Estimated Tool Implementation Cost (ETIC)	__________	
Estimated Quality Gain (EQC)	__________	
Estimated Hours of Customer Support per Project (EHCS)	__________	
Average Cost of One Hour of Customer Support (ACCS)	__________	
Metric Suites Supported		__________
McCabe	__________	
Function Points	__________	
Halstead	__________	
Features Support (FS)		__________
Extendable (tester allowed to write functions to extend tool) (E)	__________	
Database open for use by testers (DB)	__________	
Integrates with software development tools (I)	__________	
Reporting Features (RF)		__________
Summary Report automatically generated (SR)	__________	
Exportable Reports for viewing external to tool (ER)	__________	

Additional Metrics for Application to OO Software

Maximum Number of Classes (MNC)	__________
Maximum number of classes allowed in one project	
Object-Oriented Software Quality Metrics (OOSWM)	__________
Chidamber & Kemerer Metric Suite	
Weighted methods per class	__________
Depth of inheritance tree	__________
Number of children	__________
Coupling between object classes	__________
Response for a class	__________
Lie and Henry Metric Suite	
Message Passing Coupling (MPC)	__________
Data Abstraction Coupling (DAC)	__________
Henry and Kafura/Shepperd	
Information Flow Complexity (IFC)	__________
Lorenz and Kidd Metric Suite	
Number of scenarios scripts (NSS)	__________
Number of key classes (NKC)	__________
Number of messages sent by methods	__________
Number of parameters used by operation	__________
Number of subsystems (NSUB)	__________
Total number of operations + number of attributes	__________
Number of operations overridden by subclass (NOO)	__________
Number of operations added by a subclass (NOA)	__________
Specialization index (SI)	__________
Class hierarchy nesting level	__________
McCabe Object-Oriented Software Metrics	
Maximum Cyclomatic Complexity	__________
Hierarchy Quality	__________

APPENDIX F. CSMA/CD FLOW CHART



START
Read Input parameters
Initialize Variables
Scan the event list and pick an event
Type of Event
Packet departure
Packet arrival
Collision check
Transmission attempt
Update queue size and other related variables
Compute delay and push the queue forward
Schedule transmission attempt of the next packet, if any
Count the number of transmitting stations
Collision
Yes
No
Generate jamming signal
Abort all transmissions and reschedule their transmission attempts after the collision
Schedule packet departure
If more than one station is attempting then pick one at a time
Channel
Busy
Free
Schedule transmission event
Schedule collision check
Select an ID number for arriving packet
Update queue size and other variables
If queue was empty before this arrival, schedule for transmission event
Schedule the next arrival
Should the simulation be stopped?
No
Yes
Compute the output results
STOP

Figure F-1: CSMA/CD Simulation Program Flow Chart[17]

[17] Sadiku, M. and Ilyas, M., *Simulation of Local Area Networks,* Boca Raton, Florida. CRC Press, 1994, pp. 111.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Professor Bret Michael
   Naval Postgraduate School
   bmichael@cs.nps.navy.mil

4. Professor Richard Riehle
   Naval Postgraduate School
   rdriehle@nps.navy.mil

5. LT Ray Buettner
   Naval Postgraduate School
   rrbuettn@nps.navy.mil

6. Professor Mantak Shing
   Naval Postgraduate School
   shing@nps.navy.mil

7. Dr. William Bryzinski
   Software Productivity Consortium
   bryk@software.org

8. Dr. Reginald Meeson
   Institute for Defense Analyses
   meeson@ida.org

9. Mr. Robert V. Binder
   RBSC Corporation
   rbinder@rbsc.com

10. Dr. Jeffrey Voas
    Cigital
    jmvoas@cigital.com

11. Prof. Richard Kemmerer
University of California, Santa Barbara
kemm@cs.ucsb.edu

12. Mr. Neil Acantilado
SPAWAR Systems Center, San Diego
nacantil@spawar.navy.mil

13. Dr. Jeffrey Besser
SPAWAR Systems Center, San Diego
besser@spawar.navy.mil