# Metrics for Measuring the Effectiveness of Software-Testing Tools

James B. Michael, Bernard J. Bossuyt, and Byron B. Snyder
Department of Computer Science
Naval Postgraduate School
833 Dyer Rd., Monterey, CA 93943-5118, USA
{bmichael, bjbossuy, bbsnyder}@nps.navy.mil

## 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. In this paper we propose a suite of objective metrics for measuring tool characteristics, as an aid in systematically evaluating and selecting automated testing tools.*

## 1. Introduction

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?

In this paper we propose a suite of objective metrics for measuring tool characteristics, as an aid for systematically evaluating and selecting the automated testing tools that would be most appropriate for testing the system or component under test. Our suite of metrics are also intended to be used to monitor and gauge the effectiveness of specific combinations of testing tools during software development, in addition to conducting *ante* or *ex post facto* analyses.

In addition, the suite of test-tool metrics is to be used in conjunction with existing and future guidelines for conducting tools evaluations and selections. In December of 1991, a working group of software developers and tool users completed the Reference Model for Computing System-Tool Interconnections (MCSTI), known as IEEE Standard 1175; see [1] for a discussion of the MCSTI. As an offshoot of their work, they also introduced a tool-evaluation system. The system implements a set of forms which systematically guide users in gathering, organizing, and analyzing information on testing and other types of tools for developing and maintaining software. The user can view tool-dependent factors such as performance, user friendliness, and reliability, in addition to environment-dependent factors such as the cost of the tool, the tool's affect on organizational policy and procedures, and tool interaction with existing hardware and software assets of an organization. The data forms also facilitate the preference weighting, rating, and summarizing selection criteria. The process model underlying the MCSTI consists of five steps: analyzing user needs, establishing selection criteria, tool search, tool selection, and reevaluation.

## 2. Software-Quality Metrics

There is an extensive body of open-source literature on the subject of metrics for measuring the quality of software. 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, which in turn were more likely to have errors. However, software metrics have evolved well beyond the simple measures introduced in the 1960s.

### 2.1. Procedural (Traditional) Software Metrics

Metrics for traditional or procedural source code have increased in number and complexity since the first introduction of LOC. 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

This research is supported by the Space and Naval Warfare Systems Command under contract no. N00039-01-WR-D481D. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

computation of other metrics, most notably, in determining the average number of defects per thousand lines of code.

McCabe [5] first applied *cyclomatic complexity* to computer software: 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 calculated as shown in Equation 1.

$$M = V(G) = e - n + 2p \quad (1)$$

where $V(G)$ is the cyclomatic number of $G$, $e$ is the number of edges, $n$ is the number of nodes, and $p$ 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) [6] 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 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:

$$UFC = \Sigma_{1->15} \text{ (number of items of variety } i\text{) x (weight of } i\text{)} \quad (2)$$

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:

$$TCF = 0.65 + 0.01\Sigma_{1->14} F_i \quad (3)$$

where *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 [7] created a metric founded on the number of operators and operands in a program. His software-science metric (a.k.a. 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.

## 2.2. Object-Oriented Software Metrics

The most commonly cited software metrics to be computed for software with an object-oriented design are those proposed by Chidamber and Kemerer [8]. Their suite of metrics 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 can result in higher level of 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.

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 assumption here is that by separating the class into smaller classes, that the complexity of the class, and perhaps of the entire software project, can be reduced, *ceteris paribus*.

Lie and Henry [9] extended Chidamber and Kemerer's suite. 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) metric 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 development and maintenance of the software.

Henry and Kafura developed the Information Flow Complexity (IFC) metric to measure the total level of information flow of a module [10]. 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 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 [11].

$$IFC(M) = LOC(M) + [\text{fan-in}(M) \times \text{fan-out}(M)]^2 \quad (4)$$

Lorenz and Kidd [12] proposed another set of object-oriented software quality metrics. 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

## 3. Prior Work on Metrics for Software-Testing Tools

The Institute for Defense Analyses (IDA) published two survey reports on tools for testing software [2],[3]. Although the tool descriptions contained in those reports are dated, the analyses provide a historical frame of reference for the recent advances in testing tools and 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 conducted at IDA was intended to provide guidance to the U.S. Department of Defense on how to evaluate and select software-testing tools. The major conclusions of the study were that:

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

The Software Technology Support Center (STSC) at Hill AFB 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 [4]. 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/.

## 4. Proposed Suite of Metrics for Evaluating and Selecting Software-Testing Tools

Weyuker identified nine properties that complexity measures should possess [13]. Several of these properties can be applied

to other metrics too; these characteristics were considered in our formulation of metrics for evaluating and selecting software-testing tools.

Our suite of metrics for evaluating and selecting software-testing tools has the following properties: the metrics exhibit non-coarseness in that they provide different values when applied to different testing tools; the metrics 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 a metric may provide the same value when applied to different tools; and the metrics are designed to have an objective means of assessment rather than being based on subjective opinions of the evaluator.

## 4.1. Metrics for Tools that Support Testing Procedural Software

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

**4.1.1. 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.

$$HID = KMS + IFPF + ALIF + (100 - BR) \quad (5)$$

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.

**4.1.2. 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.

$$MCB = M + CB + P \quad (6)$$

where *M* (maturity) is the number of years tool (and its previous versions) have been applied in real world applications, *CB* (customer base) is the number of customers who have more than one year of experience applying the tool, and *P* (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.

**4.1.3. 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).

$$TM = AL + ICM \quad (7)$$

where *AL* (access levels) is the number of different access levels to tool information, and *ICM* (information control methods) is the sum of the different methods of controlling tool and test information.

**4.1.4. 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 following: learning time of first-time users, retainability of procedural knowledge for frequent and casual users, and operational time of frequent and casual users.

$$EU = LTFU + RFU + RCU + OTFU + OFCU \quad (8)$$

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, and *OTCU* is the average operational time for casual users.

**4.1.5. 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.

**4.1.6 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.

$$TCG = ATG + TRF \quad (9)$$

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

and *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

**4.1.7. 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.

$$TS = ART + ARTAH + ATSD - DI \quad (10)$$

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, and *DI* is the documentation inadequacy measured as the number of unsuccessful searches of documentation.

**4.1.8. 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% [14]. Vendors may also be able to provide similar statistics for their customers currently using their tools.

$$EROI = (EPG \times ETT \times ACTH) + EII - ETIC + (EQC \times EHCS \times ACCS) \quad (11)$$

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, and *ACCS* is the Average Cost of One Hour of Customer Support.

**4.1.9. Reliability (Rel).** Tool reliability is defined as the average mean time between failures.

**4.1.10. Maximum Number of Classes (MNC).** Maximum number of classes that may be included in a tool's testing project.

**4.1.11. Maximum Number of Parameters (MNP).** Maximum number of parameters that may be included in a tool's testing project.

**4.1.12. Response Time (RT).** Amount of time used to apply test case on specified size of software. RT is difficult to measure due to the varying complexity of different programs of the same size.

**4.1.13. 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

### 4.2. Metrics for Tools that Support Testing 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 Kemerer were useful in predicting the proneness of the software under test to containing faults [15]. 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.

## 5. Three Tools Selected for Use in Validating the Proposed Suite of Metrics

As a first attempt to validate our proposed suite of metrics for evaluating and selecting software-testing tools, we selected three commercial-off-the-shelf (COTS) software-testing tools against which to apply our metrics. In the following subsections, we describe each tool, discuss the setup of each tool for validation purposes, and discuss problems we encountered in exercising the tools. The tools were selected based on whether or not they support C++ and also whether or not they could be run on a Microsoft Windows platform.

### 5.1. LDRA Testbed

LDRA Testbed is a source code analysis and test coverage measurement tool. Testbed utilizes its own parsing engine. Each of its modules is integrated into an automated, software testing toolset.

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.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.

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.

### 5.2. Parasoft Testbed

For validation purposes, we used 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, black-box, 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. Programs tested were written, compiled, and executed in Microsoft Visual Studio 6.0 running on top of Microsoft Windows 2000. 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.

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.

### 5.3. Telelogic Testbed

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 (e.g., use of probes).

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.

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.

## 6. Three Versions of the Software Program Used for Validation Purposes

The validation experiments conducted were performed on three versions of discrete-event simulation programs, all of which model the same bus-type Carrier Sense Multiple Access with Collision Detection (CSMA/CD) network. The first version is a procedural program developed by Sadiku and Ilyas [16] with the modification of one line so that it could be operated on a wide range of C and C++ compilers. This version will be referred to as the *procedural version*.

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 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.

## 7. Exercising the Software-Testing Tools

### 7.1. LDRA Testbed

**7.1.1. 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, and the testing tool covered 146 out of 176 branches. 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.

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.

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)." If a Motor Industry Software Reliability Association (MISRA) code is violated, it is so annotated by the LDRA report.

**7.1.2. 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, and the testing tool covered 169 out of 196 branches. Again, in achieving 88% coverage, we only used default test settings and did not conduct additional test runs to improve our coverage.

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, 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 indicates that the program can be re-ordered to improve readability and reduce complexity.

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)."

**7.1.3. Object-Oriented.** Coverage Report – Technical difficulties with the tools prevented the generation 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.

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)."

**7.1.4. LDRA Testbed –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.

## 7.2. Parasoft Testbed

**7.2.1. Procedural.** Parasoft C++ (with integrated Code Wizard) detected 95 occurrences of eight different rule violations.

**7.2.2. Functional.** Parasoft C++ (with integrated Code Wizard) detected eighty-three occurrences of eight different rule violations 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.

**7.2.3. 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. Of the seventy-one test cases conducted, fifty passed and twenty-one reported access violation exception errors. Insure++ reported thirty-nine outstanding memory references.

**7.2.4. 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.

Results of the static analysis conducted upon the source code are reported under the "Static analysis" tab under the "Results" tab. 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.

Results of the dynamic analysis conducted on the source code are reported under the "Dynamic analysis" tab under the "Results" tab. 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:

- **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

Clicking on a test case's results will cause its branch to expand. If a test case passes, it will display the number of times it was executed and its arguments, returns, preconditions, and post-conditions.

If a test case had an error or failed, expanding its branch will display the number of times it was executed, 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.

## 7.3. Logiscope Testbed

**7.3.1. Procedural.** Telelogic's Logiscope reported 218 occurrences of fourteen different programming rule violations. If a rule is violated, it is so annotated 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.

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

**7.3.3. 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.

**7.3.4. Reporting Characteristics.** Logiscope provides its reports in HTML format, which allows for easy navigation within the reports. The report includes a separate table for each rule listing the occurrences of violations for each file. 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 that 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 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.

## 7.4. Computation of Metrics

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

**7.4.1. 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.

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

Table 1. Human-Interface Design Scores

**7.4.2. 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.

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

Table 2. Test-Case Generation Scores

**7.4.3. 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.

**7.4.4. 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.

**7.4.5. 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.

**7.4.6. 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.

**7.4.7. 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 detrimental 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.

**7.4.8. 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 & Kemerer, Lie & Henry, Lorenz & Kidd, and Henry & Kafura. Parasoft does not report any of these metrics, resulting in no score. Telelogic supports the Chidamber & Kemerer 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.

**7.4.9. 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.

**7.4.10. 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.

**7.4.11. 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.

## 8. Analysis of 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 in-

crement. 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.

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 do not pose an issue for current or prospective customers, the impact will be minimal.

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.

| | **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 |

Table 3. Summary of Tool Findings

Success was achieved in applying several of the metrics including HID, TCG, and RF. HID measurements were calculated for each testing tool based on the sub-metrics of average KMS, IFPF, ALIF, and 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.

TCG measurements also provided unique measurements for each tool. Sub-metrics measuring levels of ATG and TRF demonstrated non-coarseness, finiteness, and non-uniqueness.

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 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.

RT measurements for the three tools were all different, indicating that RT is non-coarse and finite. Although not shown in the validation results, it appears 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 FS and 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 TM and UC 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 more experiments or obtaining tool-vendor input, the latter of which is not readily available.

## 9. Conclusion

Our metrics captured differences in the three suites of software-testing tools, relative to the software system under test; the software-testing tools vary in their underlying approach, quality, and ease-of-use, among other characteristics. However, confirming evidence is needed to support our theories about the effectiveness of the tool metrics for improving the evaluation and selection of software-testing tools.

## 10. Future Directions

### 10.1. Theoretical Basis for Tool Metrics

All three anonymous reviewers commented on the lack of a theoretical foundation for our tools metrics. For instance, we express MCB as a linearly additive relationship among the variables *M*, *CB*, and *P*. However, the relationship could be nonlinear, there could be some degree of correlation among the three variables, and it may be necessary to normalize the values for each of the variables before computing MCB.

We view the development of a theoretical basis for the tool metrics as long-term research. In addition to establishing the theory for each of the metrics, it is also necessary to develop a theory of the relationship amongst the tool metrics. Two of the products of this research might be the discovery of additional types of metrics, such as time-dependent metrics for capturing the availability of software-testing tools, and what might be termed "meta metrics," that would provide information about how to interpret or apply the tool metrics.

### 10.2. Experimental Validation of Tool Metrics

Another avenue of future research is to conduct more intensive testing with the candidate tools by creating additional test cases and modifying default test settings to improve test coverage and conducting regression testing. (*N.B.*: We used the default test settings of each tool to provide a baseline for measuring tool characteristics.) One could also compare the testing tools under various operating system configurations and tool settings, or measure a tool's capability and efficiency in both measuring and improving testing coverage through modifying default settings and incorporating additional test cases. Research could also be conducted to measure a tool's ability to conduct and manage regression testing.

Moreover, one could incorporate a larger number of tool suites from different vendors with a wider spectrum of programming-language support; this would reduce the likelihood of language-specific factors affecting the research findings.

Lastly, the discrete-event simulation software program could be supplemented by case studies for which the target software has a higher degree of encapsulation, inheritance, and polymorphism. These case studies should include software systems used in real-world operational environments.

## Acknowledgements

We thank LDRA Ltd. of Wirral, United Kingdom, Parasoft Corporation of Monrovia, California, and Telelogic AB of Malmö, Sweden for their technical assistance.

## References

[1] Poston, R. M. and Sexton, M. P. Evaluating and selecting testing tools. *IEEE Software* 9, 3 (May 1992), 33-42.

[2] Youngblut, C. and Brykczynski B. An examination of selected software testing tools: 1992. IDA Paper P-2769, Inst. for Defense Analyses, Alexandria, Va., Dec. 1992.

[3] Youngblut, C. and Brykczynski, B. An examination of selected software testing tools: 1993 Supp. IDA Paper P-2925, Inst. for Defense Analyses, Alexandria, Va., Oct. 1993.

[4] Daich, G. T., Price, G., Ragland, B., and Dawood, M. Software test technologies report. Software Technology Support Center, Hill AFB, Utah, Aug. 1994.

[5] McCabe, T. J. A complexity measure. *IEEE Trans. Software Eng.* SE-2, 4 (Dec. 1976), 308-320.

[6] Dekkers, C. Demystifying function points: Let's understand some terminology. *IT Metrics Strategies*, Oct. 1998.

[7] Halstead, M. H. *Elements of Software Science*. New York: Elsevier Science, 1977.

[8] Chidamber, S. R. and Kemerer, R. F. A metrics suite for object-oriented design. *IEEE Trans. Software Eng.* 20, 6 (June 1994), 476-493.

[9] Li, W. and Henry, S. Object-oriented metrics that predict maintainability. *J. Systems and Software* 23, 2 (Nov. 1993), 111-122.

[10] Henry, S. and Kafura, D. Software structure metrics based on information flow. *IEEE Trans. Software Eng.*, SE-7, 5 (Sept. 1981), 510.

[11] Churcher, N., Shepperd, M. J., Chidamber, S., and Kemerer, C. F. Comments on "a metrics suite for object oriented design." *IEEE Trans. Software Eng.* 21, 3 (Mar. 1995), 263-265.

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

[13] Weyuker, E. J. Evaluating software complexity measures. *IEEE Trans. Software Eng.* 14, 9 (Sept. 1988), 1357-1365.

[14] *QA Quest*. The New Quality Assurance Inst., Nov. 1995.

[15] Basili, V. R., Briand, L., and Melo, W. L. A validation of object-oriented design metrics as quality indicators. Technical Report CS-TR-3443, Univ. of Md., College Park, Md., May 1995.

[16] Sadiku, M. and Ilyas, M. *Simulation of Local Area Networks*. Boca Raton, Fla: CRC Press, 1994.