AUTOMATIC TEST SOFTWARE

Even when software is developed using a rigorous discipline, it will contain a significant number of bugs. At one time, it was believed that the use of formal methods would eventually allow probably correct programs to be written, thus completely eliminating the need for testing. Today, we know that while the number of defects in a program written will be lower under certain development environments, they will still add up to a large number in a large program. There has to be testing and debugging, which can consume up to 60% of the total effort.

A study by McGibbon presents a perspective (1). He compares traditional development approach with two formal methods, VDM and Z. For a program with 30,000 source lines of code (SLOC), the traditional methods will be able to deliver software with 34 defects at an estimated life-cycle cost of $2.5 million. Using Z, the total cost would be reduced by $2.2 million, but still about 8 defects would be left. Additional cost savings can be achieved by using VDM; however, it will result in 24 defects in the delivered product. In all three cases, a substantial part of the cost is due to testing and debugging.

Finding most of the defects is a formidable task. Much of testing is still being done manually, and the process is intuitively guided. In this respect, software development lags behind hardware design and testing, where use of tools is now regarded as mandatory. The main deterrents are the learning curve of testers and the reluctance of management to commit to new tools in the middle of a project. However, as we discuss below, in a carefully implemented program, the cost/benefit ratio strongly favors the use of tools. Manual software testing is very repetitive, and thus very unattractive to software engineers. Use of appropriate tools will allow testers to use their skills at a much higher level. The software market today is extremely competitive. Many cutting-edge organizations have been able to achieve high process maturity levels, and they are delivering software with significantly lower defect densities than in the past. The average acceptable defect density declined from about 8 per thousand lines of code (KLOC) in the late 1970s to about 5 in 1990 (2). It is now believed to be below 2 per KLOC in leading-edge organizations. In the near future, the reliability expectations in the market will require all developers to greatly rely on automation in testing.

Software testing tools began appearing in the 1970s. LINT, a code checker, was part of the old Unix systems. The name was appropriately chosen in that, as Poston (3) explains, it "goes through the code and picks out all the fuzz that makes programs messy and error-prone". One of the first code instrumentors JAVS was developed by Edward Miller (who founded Software Research in 1977) in 1974 for evaluating structural coverage. In the mid-1980s computing became cheap and plentiful. Powerful software technologies such as programming environments and structured relational database systems became available, making it possible to develop tools that can capture and analyze a lot of information. Today, we see integration of capabilities such as capture–replay and comparison code coverage in the newer tools.

Recently, we have seen the emergence of a new class of tools that addresses the new programming paradigms. Memory-leak detectors emerged in 1992 and have already become indispensable in many development organizations. We now see many new tools for testing web servers and documents and for handling the Y2K problem. We can expect to see new types of tools becoming available that will automate some of the gaps in the program design–test life cycle.

Many organizations are still resisting the introduction of testing tools. Surveys suggest much of this is due to the steep learning curve faced by testers, who are very reluctant to move to new approaches when they are facing deadlines (4). Some tools have not delivered what they seemed to promise. This will change when tools are better understood. Today, no hardware engineer would think of doing a design without using SPICE- or VHDL-level simulation. No one thinks of doing manual test generation for hardware. The same will be true for software in only a few years.

In 1995 the Quality Assurance Institute conducted a benchmark study comparing manual and automated testing, involving 1750 test cases and 700 defects (5). The results are shown in Table 1. It shows that while initially the tools require some investment, they eventually result in an impressive saving in the time spent in testing.

Table 1. Manual versus Automated Testing (5)

	Time (h)		
Test Step	Manual Testing	Automated Testing	Improvement (%)
Test plan development	32	40	−25
Test case development	262	117	55
Test execution	466	23	95
Test result analyses	117	58	50
Defect tracking	117	23	80
Report creation	96	16	83
Total hours	1090	277	75

TERMINOLOGY

The following are the important terms used in the software testing literature.

Failure. A departure of the system behavior from user requirements during execution.

Defect (or Fault). An error in system implementation that can cause a failure during execution. A defect will cause a failure only when the erroneous code is executed and the effect is propagated to the output.

Defect Density. Usually measured as the number of defects per thousand source lines of code (KSLOC).

Failure Intensity. The expected number of failures per unit time.

Mean Time to Failure (MTTF). The expected duration between two successive failures. It is the reciprocal of the failure intensity.

Operational Profile. To be able to estimate operational reliability, testing must be done in accordance with the operational profile. The *profile* is the set of disjoint actions (operations) that a program may perform, and their probabilities of occurrence. The probabilities that occur in actual operation specify the *operational profile.* Obtaining an operational profile requires dividing the input space into sufficiently small leaf partitions and then estimating the probabilities associated with each leaf partition. A subspace with high probability may need to be further divided into smaller subspaces.

SOFTWARE DEVELOPMENT AND TEST PHASES

A competitive and mature software development organization targets a high reliability objective from the very beginning of software development. Generally, the software life cycle is divided into the following phases. As we will see later, different testing-related tools may be required for different phases.

A. *Requirements and Definition.* The developing organization interacts with the customer organization to specify the software system to be built. Ideally, the requirements should define the system completely and unambiguously. In actual practice, there is often a need to do corrective revisions during software development. A review or inspection during this phase is generally done by the design team to identify conflicting or missing requirements. A significant number of errors can be detected by this process. A change in the requirements in the later phases can cause increased defect density.

B. *Design.* The system is specified as an interconnection of units, such that each unit is well defined and can be developed and tested independently. The design is reviewed to recognize errors.

C. *Coding.* The actual program for each unit is written, generally in a high-level language such as C or C++. Occasionally, assembly-level implementation may be required for high performance or for implementing input/output operations. The code is inspected by analyzing the code (or specification) in a team meeting to identify errors.

D. *Testing.* This phase is a critical part of the quest for high reliability and can take 30% to 60% of the entire development time. It is generally divided into these subphases.

   Unit Test. Each unit is tested separately, and changes are made to remove the defects found. Since each unit is relatively small and can be tested independently, it can be exercised much more thoroughly than a large program.

   Integration Testing. During integration, the units are gradually assembled, and partially assembled subsystems are tested. Testing subsystems allows the interface among modules to be tested. By incrementally adding units to a subsystem, the unit responsible for a failure can be identified more easily.

   System Testing. The system as a whole is exercised during system testing. Debugging is continued until some exit criterion is satisfied. The objective of this phase is to find defects as fast as possible. In general the input mix may not represent what would be encountered during actual operation.

   Acceptance Testing. The purpose of this test phase is to assess the system reliability and performance in the operational environment. This requires collecting (or estimating) information about how the actual users would use the system. This is also called alpha testing. It is often followed by beta testing, which involves actual use by the users.

   Regression Testing. When significant additions or modifications are made to an existing version, regression testing is done on the new, or *build,* version to ensure that it still works and has not *regressed* to lower reliability.

E. *Operational Use.* Once the software developer has determined that an appropriate reliability criterion is satisfied, the software is released. Any bugs reported by the users are recorded but are not fixed until the next release.

SOFTWARE TEST METHODOLOGY

To test a program, a number of inputs are applied and the program response is observed. If the response is different from what was expected, the program has at least one defect. Testing can have one of two separate objectives. During debugging, the aim is to increase the reliability as fast as possible, by finding faults as quickly as possible. On the other hand, during certification, the objective is to assess the reliability; thus the fault-finding rate should be representative of actual operation. The test generation approaches can be divided into the following classes:

A. *Black-Box (or Functional) Testing.* When test generation is done by only considering the input/output description of the software, nothing about the implementation of the software is assumed to be known. This is the most common form of testing.

B. *White-Box (or Structural) Testing.* In this approach the actual implementation is used to generate the tests. While test generation using the white-box approach is not common, evaluation of test effectiveness often requires use of structural information.

COVERAGE MEASURES

The extent to which a program has been exercised can be evaluated by measuring software *test coverage* (6). Test coverage in software is measured in terms of structural or dataflow units that have been exercised. These units can be statements (or blocks), branches, and so on, as defined below. Some popular coverage measures are often referred to by using a compact notation, these are given in parentheses.

Statement Coverage (C0). The fraction of the total number of statements that have been executed by the test data.

Branch Coverage (C1). The fraction of the total number of branches that have been executed by the test data.

C-Use Coverage. The fraction of the total number of computation uses (c uses) that have been covered during testing. A c-use pair includes two points in the program: a point where the value of a variable is defined or modified, followed by a point where it is used for computation (without the variable being modified along the path).

P-Use Coverage. The fraction of the total number of predicate uses (p uses) that have been covered during testing. A p-use pair includes two points in the program: a point where the value of a variable is defined or modified, followed by a point that is a destination of a branching statement where it is used as a predicate (without modifications to the variable along the path). It has been shown that if all paths in the program have been exercised, then all p uses must have been covered. This means that the all-paths coverage requirement is stronger than the all-p-use. Similarly, all-p-use coverage implies all-branches coverage, and all-branches coverage implies all-instructions coverage. This is termed the *subsumption hierarchy.*

Module Coverage (S0). The fraction of the total number of modules that have been called during testing. A *module* is a separately invocable element of a software system, sometimes also called a procedure, function, or program.

Call-pair Coverage (S1). The fraction of the total number of call pairs that have been used during testing. A *call pair* is a connection between two functions in which one function calls (references) the other function.

Path Coverage. The fraction of the total number of paths that have been used during testing. A *path* is any sequence of branches taken from the beginning to the end of a program. To achieve 100% path coverage, all permutations of paths must be executed.

Tools for different phases are examined below. Some tools are applicable to multiple phases. Some of the types of tools are now widely used; others have just started to emerge.

REQUIREMENTS-PHASE TOOLS

Requirement Recorder/Verifier

Requirements can be recorded informally in a natural language such as English or formally using Z, LOTOS, etc. Use of formal methods results in a more thorough recording of requirements. The requirement information needs to be unambiguous, consistent, and complete. A term or an expression is *unambiguous* if it has one and only one definition. A requirements specification is *consistent* if each term is used in the same manner for each occurrence. *Completeness* implies the presence of all needed statements and of all required components for each statement. The requirement verifiers can automatically check for ambiguity, inconsistency, and completeness of statements. However, they cannot determine that the set of requirement statements is complete. This would require review by human testers. A requirements recorder may also assist in specification-based test case generation.

Test Case Generation

Automatic test case generation can be an extremely important part of achieving high-reliability software. Manual test case generation is a slow and labor intensive process and may be insufficient if not done carefully. Arbitrarily generated tests can find defects with high testability relatively easily; however, these tests can become ineffective as testing progresses. Specification-based test generation can ensure that the different test cases cover at least some different functionality by partitioning the functionality and probing the portions. Either the input space or the state space may be partitioned. Poston (3) classifies the strategies used as active-driven (to test for missing actions), data-driven, logic-driven, event-driven, and state-driven. Both Validator (Aonix) and T-VEC (T-VEC) include specification verification and test case generation.

Orthogonal to the test generation strategy is question of test vector distribution. The distribution may be chosen to conform with the operational profile, so that the tests replicate the normal operation. On the other hand, the strategy, at each step, may choose to probe a functionality that has been relatively untouched by testing so far. The second approach may be implemented in the form of antirandom testing (7). A combinatorial-design-based test generation can significantly reduce the number of combinations to be considered. This is the approach used in AETG (Bellcore) (8).

It is also possible to generate tests using the software implementation formulation. Some tools can use this approach, termed "white-box" testing. Such test generation can require enormous amounts of computation, and thus should be considered only for branches, p uses, and the like that are very hard to test otherwise. Beizer has called such testing "kiddie testing at its worst." Such tests cannot detect missing functions (3).

PROGRAMMING-PHASE TOOLS

These are often called "static" tools, because they do not involve actual execution of the software.

Metric Evaluators

Many metrics have been used to evaluate aspects of the complexity of programs. They include lines of code, number of modules, operands, operators, and data/control flow measures. The belief is that if a module is more complex, it is more fault-prone and thus deserves special attention. It has been shown that many metrics are strongly correlated with the number of lines of code, and may not provide any further information (9). Still, when the resources and time are limited, it may be a good strategy to identify the fault-prone modules. Poston regards such tools as "nice" but not essential.

Code Checkers

These are also static tools like metric evaluators. These tools look for violations of good programming practices to generate warnings. They can identify misplaced pointers, uninitialized variables, and nonconformance with coding standards. STW/Advisor (Software Research) includes both code checking and metric evaluation.

Inspection-Based Error Estimation

A design document or code can be inspected. Many defects can often be detected simply by inspection. If separate teams or individuals do inspection independently, it amounts to sampling the defects present. Statistical methods are available that can be used to obtain a preliminary estimate of the remaining number of bugs remaining (10).

TESTING-PHASE TOOLS

These tools were the earliest to appear and are now widely used. They are often termed “dynamic” because they involve actual execution of the software using the test cases selected and evaluation of test thoroughness.

Capture–Playback Tools

These are somewhat like a VCR, or perhaps more closely like recording and running spreadsheet macros. Older capture–playback tools worked at the bit-map level. Modern tools can capture and replay information at the bit-map, widget, object, or control level. Information captured can be edited to replace hard-coded values and pathnames to make them more general by passing setting environment variables and passing parameter values. One can build a library of small test scripts, which can be combined to obtain different test sequences. A test sequence can be implemented by using a state table as a driver.

An alternative is to have data-driven scripts that input data as well as parameters and environment variables. Using appropriate data values, the same test scripts can be made to cover different functionalities of the program. The data files can also contain the expected results for specific test cases, such as success or failure. Most capture–playback tools today incorporate a comparator, which compares the expected and actual outputs. QA Partner (Seague) and WinRunner/Xrunner (Mercury Interactive) are examples of this class of tools.

Memory-Leak Detectors

Modern programming practices use dynamic memory allocation. If a program fails to deallocate memory that is no longer being used, it keeps on reserving more and more of the memory to itself, until eventually it runs out of memory. Such memory leaks can be detected by tools such as Bounds-Checker (Relational Software) or Purify (Purify).

Test Harnesses

Software under test needs to interface with a capture–replay tool as well as a database system and perhaps with other systems also. These interfaces also need to be tested. Such a test execution environment is termed a test harness. It may include “stubs” to stimulate missing parts. In the past, test har-

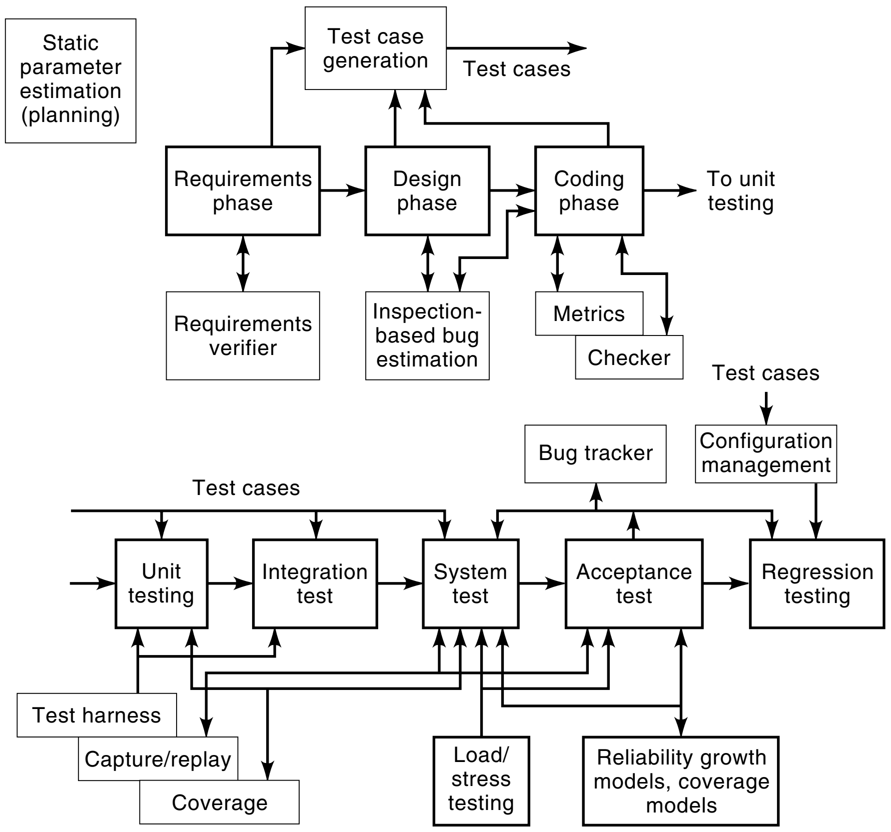

Figure 1. Software development and test phases and testing and reliability tools.

nesses have been custom-built. Some test harness generators, such as Cantata (Information Processing), have recently become available.

Coverage Analyzers

It is impossible to test a program exhaustively. The testers must decide if a program has been exercised sufficiently thoroughly. One way is to use a coverage analyzer, which will keep track of the fraction of all structural or data-flow units that have been exercised. It has been shown that coverage measures are approximately linearly correlated with the defect coverage (6).

Most analyzers are intrusive. They "instrument" the code by inserting test probes in the software before it is compiled. Instrumenting affects the performance slightly. Nonintrusive analyzers are a much more expensive alternative; they collect information using a separate hardware processor.

Statement coverage is not a rigorous measure even with 100% coverage: the residual defect density can still be high. Branch coverage is a popular remedy. Sometimes a threshold value, say 85% branch coverage, is used. Pure coverage is stricter than branch coverage and is suitable for high-reliability programs. Module coverage and call-pair coverage are common system-level coverage measures. At the present time use path coverage is feasible only if its definition is revised to reduce the total path count. GCT (Testing Foundations) and ATAC (Bellcore) are coverage analyzers.

Load/Performance Testers

These tools allow stress testing of client/server applications, which are often expected to work correctly under high loads. SQA LoadTest (SQA) allows stress testing of Windows client/server applications; Final Exam Internet Load Test (Platinum) is specifically for web applications.

Bug Trackers

A bug tracker records the status of each bug found. Depending on the strategy used, a bug may or may not be fixed immediately after it is found. A bug-tracker is basically a database tool.

Reliability Growth Modeling Tools

As defects are found and removed, the reliability of the program increases. This is manifested by a gradual decline in the defect finding rate. A wealth of methods is available that use software reliability growth models (SRGMs). Several SRGM tools are available that have these features (11):

1. Preprocessing or smoothing of data
2. Estimating parameters of a selected SRGM
3. Answering queries such as how much longer the software needs to be tested

SMERFS (NSWRC) is a popular SRGM tool. ROBUST (CSU) allows parameters of SRGMs to be estimated even before testing begins, which can be useful for preliminary resource planning.

Coverage-Based Reliability Tools

Recently, a model describing defect coverage and test coverage has been proposed and validated. The model tends to fit the data quite closely and can yield very stable estimates of the number of residual defects (12). ROBUST (CSU) allows coverage to be used as the stopping rule criterion. It also allows stable estimation of the number of residual defects (12).

Table 2. TestWorks Quality Index (13)

		Requirement						My Score
No.	Evaluation Factor	50 Points	60 Points	70 Points	80 Points	90 Points	100 Points	on This Factor
F1	Cumulative C1 for all tests	>25%	>40%	>60%	>85%	>90%	>95%	
F2	Cumulative S1 for all tests	>50%	>65%	>80%	90%	95%	95%	
F3	Functions with cyclomatic complexity $E(n) < 20$	<25%	>25%	>50%	>75%	>90%	>95%	
F4	Functions with "clean" static analysis	<20%	>20%	>30%	>40%	>50%	>60%	
F5	Last pass/fail percentage	>25%	<25%	>50%	>75%	>90%	>95%	
F6	Total number of test cases per KLOC	<10	>10	>15	>20	>30	>40	
F7	Calling tree aspect ratio (width/height)	>1.0	<1.25	<1.5	<1.75	<2.0	>2.0	
F8	Number of open criticality-1 Defects per KLOC	>5	<5	<3	<2	<1	<0.5	
F9	Functions for which path coverage is performed	<1%	>2%	>5%	>10%	>15%	>25%	
F10	Cost per defect	>$100K	>$50K	>$25K	>$10K	>$1K	<$1K	
	Total points	→	→	→	→	→	→	

IDENTIFYING THE TOOLS NEEDED

Software testing tools can be expensive. The cost to license a tool can be just a fraction of the overall cost. The testers need to understand the tools and become familiar with them. The use of the tools needs to be incorporated in the process.

Poston (3) regards these as the essential tools at most development organizations; he terms them the "Big 3" tools:

1. Requirement recorder and test case generator
2. Test execution tool
3. Test evaluation tool

He refers to some of the other tools as "nice to have" and considers structure-based test generation tools to be useless.

Not all projects need sophisticated tools. Many can significantly benefit by using some of the simpler tools. One good approach to identifying the tools needed is to consider the quality required in the final project. A measure of quality called TestWorks Quality Index has been defined by Software Research (13). It is composed of 10 additive factors as given in Table 2.

For example, a "quick and dirty" (but still useful) order tracker may have the quality index calculated as in Table 3. On the other hand, a bedside cardiac monitor may be required to have a much higher quality index, as calculated in Table 4. It will require reliance on more powerful tools to achieve higher quality.

SOURCES OF INFORMATION

Here major sources of detailed information on software testing related tools are listed.

Web Sites

1. Testing Tools Supplier List, http://www.stlabs.com/MARICK/faqs/tools.htm, includes information and links to a very detailed list of tools, classified into test design tools, GUI test drivers and capture/replay tools, load and performance tools, non-GUI test drivers and test managers, other test implementation tools, test evaluation tools, static analysis tools, and miscellaneous tools.
2. Testing and Test Management Tools, http://www.methods-tools.com/tools/frames_testing.html, another detailed list of tools.
3. RST Hotlist, http://www.rstcorp.com/hotlist.html.
4. STORM Software Testing On-line Resources, http://www.mtsu.edu/~storm/.
5. SR/Institute's Software Quality HotList, http://www.soft.com/Institute/HotList/index.html.
6. Software Testing Hotlist, http://www.io.com/~wazmo/qa.html.
7. Papers by Cem Kaner, http://www.kaner.com/writing.htm.
8. The Testers' Network, http://www.stlabs.com/testnet.htm.

Table 3. A "Quick and Dirty" Order Tracker (13)

Cumulative C1 (branch coverage) value for all tests	75
Cumulative S1 (call-pair coverage) value for all tests	70
Percentage of functions with $E(n) < 20$	50
Percentage of functions with clean static analysis	50
Last pass/fail percentage	80
Total number of test cases per KLOC	50
Calling tree aspect ratio (width/height)	60
Current number of open defects per KLOC	50
Percentage of functions for which path coverage is performed	50
Cost per defect	100
Total points scored	535
TestWorks Quality Index	53.5

Table 4. Bedside Cardiac Monitor (13)

Cumulative C1 (branch coverage) value for all tests	100
Cumulative S1 (call-pair coverage) value for all tests	100
Percentage of functions with $E(n) < 20$	80
Percentage of functions with clean static analysis	80
Last pass/fail percentage	90
Total number of test cases per KLOC	85
Calling tree aspect ratio (width/height)	60
Current number of open defects per KLOC	95
Percentage of functions for which path coverage is performed	60
Cost per defect	50
Total points scored	800
TestWorks Quality Index	80

Books

1. D. Hoffman and P. Strooper, *Software Design, Automated Testing, and Maintenance: A Practical Approach,* London: International Thomson Computer Press, 1995.
2. Graham Titterington, *Ovum Evaluates: Software Testing Tools,* London: Ovum Limited, 1998.
3. R. M. Poston, *Automating Specification-Based Software Testing,* Los Alamitos, CA: IEEE Computer Society Press, 1996.
4. L. G. Hayes, *The Automated Testing Handbook,* Richardson, TX: Software Testing Institute, 1995.

Conferences

1. IEEE High-Assurance Systems Engineering Workshop
2. International Conference on Computer Safety, Reliability and Security
3. International Conference on Software Maintenance
4. Metrics—International Symposium on Software Metrics
5. ISSRE—International Symposium on Software Reliability Engineering

BIBLIOGRAPHY

1. T. McGibbon, An analysis of two formal methods, VDM and Z [Online], 1997. Available www: http//www.dacs.dtic.mil
2. The quality imperative, *Business Week,* special bonus issue, Fall 1991.
3. R. Poston, *A Guided Tour of Software Testing Tools,* San Francisco: Aonix, 1998.
4. Results from August's survey on automated testing, quality tree [Online], 1997. Available www:http://www/qualitytree.com/survey/august/results.htm
5. *QA Quest, The Newsletter of the Quality Assurance Institute,* Nov. 1995.
6. Y. K. Malaiya et al., The relationship between test coverage and reliability, *Proc. Int. Symp. Software Reliability Eng.,* 1994, pp. 186–195.
7. H. Yin, Z. Lebne-Dengal, and Y. K. Malaiya, Automatic test generation using checkpoint encoding and antirandom testing, *Proc. Int. Symp. Software Reliability Eng.,* 1997, pp. 84–95.
8. D. M. Cohen et al., The AETG system: An approach to testing based on combinatorial design, *IEEE Trans. Softw. Eng.,* **23**: 437–444, 1997.
9. J. Rosenberg, Some misconceptions about lines of code, *Proc. Int. Software Metrics Symp.,* 1997, pp. 137–142.
10. N. B. Ebrahimi, On the statistical analysis of the number of errors remaining in a software design document after inspection, *IEEE Trans. Softw. Eng.,* **23**: 529–532, 1997.
11. M. R. Lyu (ed.), *Software Reliability Engineering,* New York: McGraw-Hill, 1996.
12. Y. K. Malaiya and J. Denton, Estimating defect density using test coverage, Colorado State University Tech. Report CS-98-104, 1998.
13. TestWorks Quality Index: Overview, software research application note, San Francisco, CA: Software Research, Inc., 1998.

YASHWANT K. MALAIYA
Colorado State University