Software debugging, testing, and verification

by B. Hailpern
P. Santhanam

In commercial software development organizations, increased complexity of products, shortened development cycles, and higher customer expectations of quality have placed a major responsibility on the areas of software debugging, testing, and verification. As this issue of the* IBM Systems Journal *illustrates, there are exciting improvements in the underlying technology on all three fronts. However, we observe that due to the informal nature of software development as a whole, the prevalent practices in the industry are still immature, even in areas where improved technology exists. In addition, tools that incorporate the more advanced aspects of this technology are not ready for large-scale commercial use. Hence there is reason to hope for significant improvements in this area over the next several years.

Although the complexity and scope of software has increased tremendously over the past decades, advances in software engineering techniques for producing the software have been only moderate, at best. Software development has remained primarily a labor-intensive effort and thus subject to human limitations. As Frederick Brooks explained over a quarter of a century ago,[1] there is a big difference between an isolated program created by a lone programmer and a programming systems product. A programming systems product "can be run, tested, repaired, and extended by anybody . . . in many operating environments, for many sets of data" and forms a part of "a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks." Brooks asserted a nine-fold increase in cost to develop a programming system product from an isolated program (see Figure 1).

With the advent of the Internet and the World Wide Web, the problems that were recognized a quarter century ago as having "no silver bullet" for the solution[1] have been magnified. The challenges of designing and testing distributed computing systems, with distributed data and Web services, with the need for coexistence of heterogeneous platforms, unpredictable run-time environments, and so on, make the already difficult problem even harder.

A key ingredient that contributes to a reliable programming systems product is the assurance that the program will perform satisfactorily in terms of its functional and nonfunctional specifications within the expected deployment environments. In a typical commercial development organization, the cost of providing this assurance via appropriate debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost. Thus we should consider what is involved in these activities that make them so challenging and so expensive.

Since one of the goals of this special issue of the *IBM Systems Journal* is to be accessible to the students of software engineering at large, we define relevant terminology and its implications (we include formal no-

©Copyright 2002 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

 0018-8670/02/$5.00 © 2002 IBM

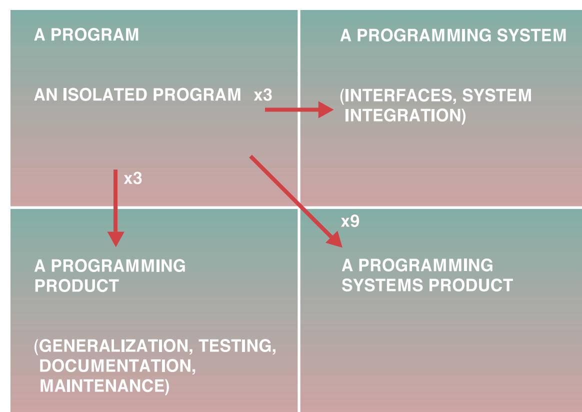
tation for this terminology, but it is not essential for the basic understanding of problem definition). Note that the terms "debugging," "testing," and "verification" are not mutually exclusive activities, especially in everyday practice. The definitions draw distinctions, but the boundaries may actually be fuzzy. We begin with a software program written in a programming language (let P be the program written in language L). The program is expected to satisfy a set of specifications, where those specifications are written in a specification language (call the set of specifications $\Phi = \{\phi_1, \phi_2, \phi_3, \ldots, \phi_n\}$ and the specification language Λ). In most real-world cases, the specification language (Λ) is the natural language of the development team (i.e., English, Spanish, etc.).

Debugging: The process of debugging involves analyzing and possibly extending (with debugging statements) the given program that does not meet the specifications in order to find a new program that is close to the original and does satisfy the specifications (given specifications Φ and a program P, not satisfying some $\phi_k \in \Phi$, find a program P′ "close" to P that does satisfy ϕ_k). Thus it is the process of "diagnosing the precise nature of a known error and then correcting it."[2]

Verification: Given a program and a set of specifications, show that the program satisfies those specifications (given P and a set of specifications $\Phi = \{\phi_1, \phi_2, \phi_3, \ldots, \phi_n\}$, show that P satisfies Φ). Thus, verification is the process of proving or demonstrating that the program correctly satisfies the specifications.[2] Notice that we use the term verification in the sense of "functional correctness," which is *different* from the typical discussion of verification activities discussed in some software engineering literature,[3,4] where it applies to ensuring that "each step of the development process correctly echoes the intentions of the immediately preceding step."

Testing: Whereas verification proves conformance with a specification, testing finds cases where a program does not meet its specification (given specifications Φ and a program P, find as many of $\phi_1, \phi_2, \phi_3, \ldots, \phi_p \in \Phi$ not satisfied by P). Based on this definition, any activity that exposes the program behavior violating a specification can be called testing. In this context, activities such as design reviews, code inspections, and static analysis of source code can all be called testing, even though code is not being executed in the process of finding the error or unexpected behavior. These activities are sometimes referred to as "static testing."[5] Of course, execution

Figure 1 Evolution of a programming systems product



F. P. Brooks, *The Mythical Man-Month*, p.5, Figure 1-1. © 1995, 1975 Addison Wesley Longman, Inc. Reproduced by permission of the author and Pearson Education, Inc.

of code by invoking specific test cases targeting specific functionality (using, for example, regression test suites) is a major part of testing.

Validation: Validation is the process of evaluating software, at the end of the development process, to ensure compliance with requirements. Note that the verification community also uses the term validation to differentiate formal functional verification from extensive testing of a program against its specifications.

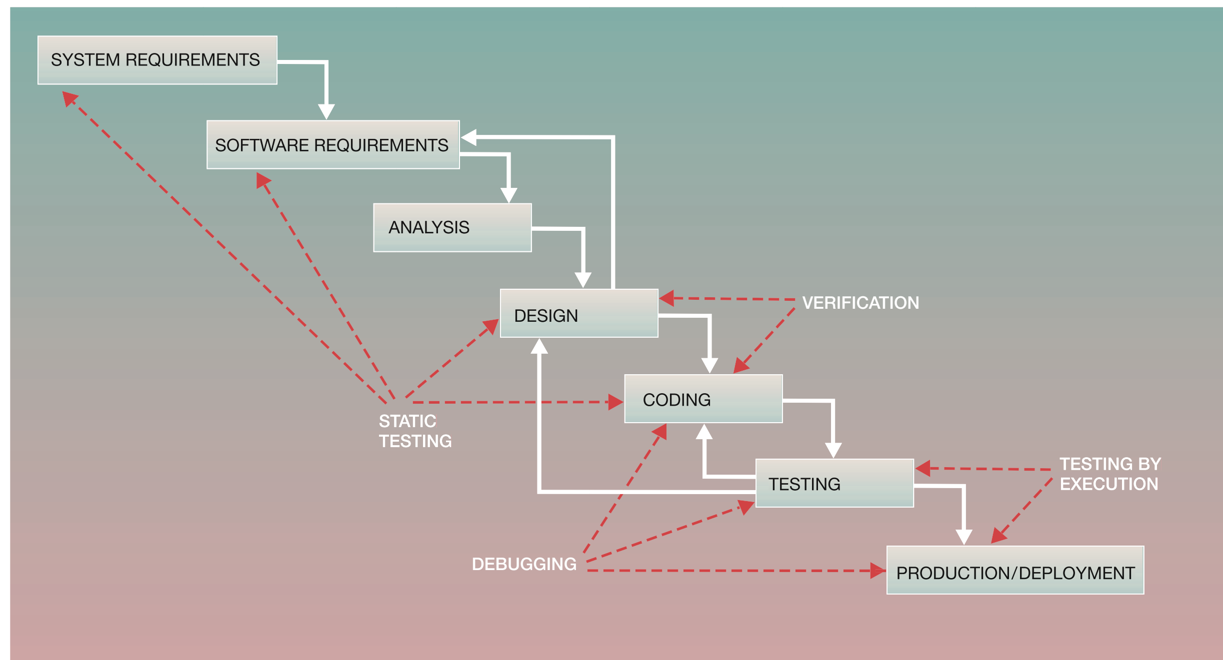
Defect: Each occurrence of the program design or the program code that fails to meet a specification is a defect (bug).[6]

Debugging, testing, and verification mapped to the software life cycle

In a typical software development process, irrespective of the specific development model followed (i.e., waterfall, iterative, spiral, etc.[5,7]), certain basic activities are required for a successful execution of a project, as illustrated in Figure 2. In this context, it is useful to know the specific roles played by debugging, testing, and verification.

Debugging. The purpose of debugging is to locate and fix the offending code responsible for a symptom violating a known specification. Debugging typically happens during three activities in software development, and the level of granularity of the analysis required for locating the defect differs in these three.

Figure 2 The activities that involve debugging, testing, and verification in a typical software development process

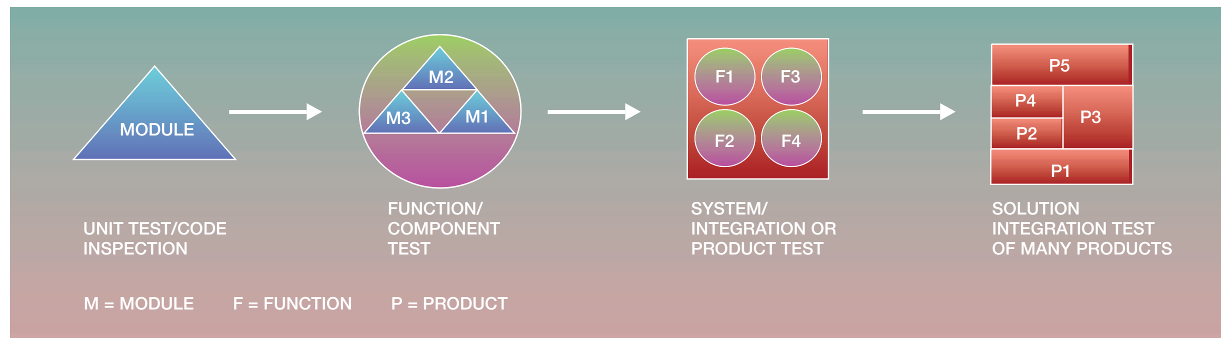


The first is during the coding process, when the programmer translates the design into an executable code. During this process the errors made by the programmer in writing the code can lead to defects that need to be quickly detected and fixed before the code goes to the next stages of development. Most often, the developer also performs unit testing to expose any defects at the module or component level. The second place for debugging is during the later stages of testing, involving multiple components or a complete system, when unexpected behavior such as wrong return codes or abnormal program termination ("abends") may be found. A certain amount of debugging of the test execution is necessary to conclude that the program under test is the cause of the unexpected behavior and not the result of a bad test case due to incorrect specification, inappropriate data, or changes in functional specification between different versions of the system. Once the defect is confirmed, debugging of the program follows and the misbehaving component and the required fix are determined. The third place for debugging is in production or deployment, when the software under test faces real operational conditions. Some undesirable aspects of software behavior, such as inadequate performance under a severe workload or unsatisfactory recovery from a failure, get exposed at this stage and the offending code needs to be found and fixed before large-scale deployment. This process may also be called "problem determination," due to the enlarged scope of the analysis required before the defect can be localized.

Verification. In order to verify the functional correctness of a program, one needs to capture the model of the behavior of the program in a formal language or use the program itself. In most commercial software development organizations, there is often no formal specification of the program under development. Formal verification[8] is used routinely by only small pockets of the industrial software community, particularly in the areas of protocol verification and embedded systems. Where verification is practiced, the formal specifications of the system design (derived from the requirements) are compared to the functions that the code actually computes. The goal is to show that the code implements the specifications.

Testing. Testing is clearly a necessary area for software validation. Typically, prior to coding the program, design reviews and code inspections are done

Figure 3 Typical stages of testing within IBM



as part of the static testing effort.[9] Once the code is written, various other static analysis methods based on source code can be applied.[5] The various kinds and stages of testing that target the different levels of integration and the various modes of software failures are discussed in a wide body of literature.[2,10,11] The testing done at later stages (e.g., external function tests, system tests, etc., as shown in Figure 3) is "black box" testing, based on external specifications, and hence does not involve the understanding of the detailed code implementations. Typically, system testing targets key aspects of the product, such as recovery, security, stress, performance, hardware configurations, software configurations, etc. Testing during production and deployment typically involves some level of customer-acceptance criteria. Many software companies have defined prerelease "beta" programs with customers to accomplish this.

Current state of technology and practice

In commercial hardware development, it is a common practice to capture the requirements and specifications in a formal manner and use them extensively in the development and testing of the products. The cost of bad or imprecise specification is high and the consequences are severe. In contrast, software poses feasibility challenges for the capture and use of such information. A software development organization typically faces:

- Ever-changing requirements (which, in many cases, are never written down)
- Software engineers' lack of skills in formal techniques
- Enormous time pressure in bringing product to market
- Belief in the malleability of software—that whatever is produced can be repaired or enhanced later

Consequently, in most software organizations, neither the requirements nor the resulting specifications are documented in any formal manner. Even if written once, the documents are not kept up to date as the software evolves (the required manual effort is too burdensome in an already busy schedule). Specifications captured in natural languages are not easily amenable to machine processing.

Should one wish to go beyond fuzzy specifications written in a natural language, there is a long history of many intellectually interesting models and techniques[12–14] that have been devised to formally describe and prove the correctness of software: Hoare-style assertions, Petri nets, communicating sequential processes, temporal logic, algebraic systems, finite state specifications, model checking, and interval logics. A key aspect of formal modeling is that the level of detail needed to capture the adequate aspects of the underlying software program can be overwhelming. If all of the details contained in the program are necessary to produce the specification or test cases, then the model may well be at least as large as the program, thus lessening its attractiveness to the software engineers. For example, there have been several attempts to model software programs as finite state machines (FSMs). While FSMs have been successful in the context of embedded systems and protocol verification, state-based representation of software leads to explosion in the number of states very quickly. This explosion is a direct result of software constructs such as unbounded data structures, unbounded message queues, the asynchronous nature of different software processes (without a global syn-

chronizing clock), and so on. In order to be relevant and manageable, software models have to use techniques such as symbolic algorithms, partial order reduction, compositional reasoning, abstraction, symmetry, and induction.[12]

There are many formal languages, such as Z, SDL (Specification and Description Language), and Promela, that can be used to capture specifications, but they are used consistently by only small pockets of the industrial software community. Unlike other areas of computer science and engineering, software debugging, testing, and verification techniques have evolved as practitioners' collections of tricks, rather than as a well-accepted set of theories or practices.

Debugging. As is well known among software engineers, most of the effort in debugging involves locating the defects. Debugging is done at the smallest level of granularity during the coding process. In the early years of software development, defects that escaped code reviews were found by compilation and execution. Through a painful process (such as inserting print statements for the known outputs at the appropriate places in the program), a programmer could locate the exact location of the error and find a suitable fix.

Even today, debugging remains very much an art. Much of the computer science community has largely ignored the debugging problem.[15] Eisenstadt[16] studied 59 anecdotal debugging experiences and his conclusions were as follows: Just over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools. The predominant techniques for finding bugs were data gathering (e.g., print statements) and hand simulation. The two biggest causes of bugs were memory overwrites and defects in vendor-supplied hardware or software.

To help software engineers in debugging the program during the coding process, many new approaches have been proposed and many commercial debugging environments are available. Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation. One area that has caught the imagination of the industry is the visualization of the necessary underlying programming constructs as a means to analyze a program.[17,18] There is also considerable work in trying to automate the debugging process through program slicing.[19]

When the testing of software results in a failure, and analysis indicates that the test case is not the source of the problem, debugging of the program follows and the required fix is determined. Debugging during testing still remains manual, by and large, despite advances in test execution technology. There is a clear need for a stronger (automatic) link between the software design (what the code is intended to do), test creation (what the test is trying to check), and test execution (what is actually tested) processes in order to minimize the difficulty in identifying the offending code when a test case fails. Debugging during production or after deployment is very complicated. Short of using some advanced problem determination techniques for locating the specific defect or deficiency that led to the unexpected behavior, this debugging can be painful, time consuming, and very expensive. The problems are exacerbated when problem determination involves multiple interacting software products. As debugging moves away from actual programming of the source code (for example, in system test, or even later in customer beta test), problem determination becomes even more manual and time-consuming.

Verification. As discussed earlier, in order to verify the functional correctness of a program, one needs to capture the specifications for the program in a formal manner. This is difficult to do, because the details in even small systems are subtle and the expertise required to formally describe these details is great. One alternative to capturing a full formal specification is to formalize only some properties (such as the correctness of its synchronization skeleton) and verify these by abstracting away details of the program. For network protocols, reactive systems, and microcontroller systems, the specification of the problem is relatively small (either because the protocols are layered with well-defined assumptions, inputs, and outputs, or because the size of the program or the generality of the implementation is restricted) and hence tractable by automatic or semiautomatic systems. There is also a community that builds a model representing the software requirements and design[12,13] and verifies that the model satisfies the program requirements. However, this does not assure that the implemented code satisfies the property, since there is no formal link between the model and the implementation (that is, the program is not derived or created from the model).

Historically, software verification has had little impact on the real world of software development. Despite the plethora of specification and verification technologies, the problem has been in applying these techniques and theories to full-scale, real-world programs. Any fully detailed specification must, by its very nature, be as complex as the actual program. Any simplification or abstraction may hide details that may be critical to the correct operation of the program. Similarly, any proof system that can automatically verify a real program must be able to handle very complex logical analyses, some of which are formally undecidable. The use of complex theorem-proving systems also requires a high skill level and does not scale to large programs. The human factor also enters into the equation: crafting a correct specification (especially one using an obscure formal system) is often much more difficult than writing the program to be proved (even one written in an obscure programming language).[20] To date, success in program verification has come in restricted domains where either the state space of the problem is constrained or only a portion of the program is actually verified. General theorem provers, model checkers, state machine analyzers, and tools customized to particular applications have all been used to prove such systems.

Any proof system that can automatically verify a real program must be able to handle very complex logical analyses.

Testing. Dijkstra's criticism,[21] "Program testing can be used to show the presence of bugs, but never to show their absence" is well known. From his point of view, any amount of testing represents only a small sampling of all possible computations and is therefore never adequate to assure the expected behavior of the program under all possible conditions. He asserted that "the extent to which the program correctness can be established is not purely a function of the program's external specifications and behavior but it depends critically upon its internal structure." However, testing has become the preferred process by which software is shown, in some sense, to satisfy its requirements. This is primarily because no other approach based on more formal methods comes close to giving the scalability and satisfying the intuitive "coverage" needs of a software engineer. Hamlet[22] linked good testing to the measurement of the dependability of the tested software, in some statistical sense. The absence or presence of failures as exposed by testing alone does not measure the dependability of the software, unless there is some way to quantify the testing properties to be certain that adequate dimensions of testing, which include the testability of the target software, were covered. Test planning techniques[10,11] based on partitioning of the functionality, data, end-user operational profiles,[23] and so on are very useful and popular in testing research and among practitioners. Many of the current technologies in testing are based on these ideas.

Test metrics. As discussed earlier, testing is, indeed, a sampling of the program execution space. Consequently, the natural question arises: when do we stop testing? Given that we cannot really show that there are no more errors in the program, we can only use heuristic arguments based on thoroughness and sophistication of testing effort and trends in the resulting discovery of defects to argue the plausibility of the lower risk of remaining defects. Examples of metrics used[24] during the testing process that target defect discovery and code size are: product and release size over time, defect discovery rate over time, defect backlog over time, and so on. Some practical metrics that characterize the testing process are: test progress over time (planned, attempted, actual), percentage of test cases attempted, etc. In some organizations, the use of test coverage metrics (e.g., code coverage) are used to describe the thoroughness of the testing effort. The use of Orthogonal Defect Classification (ODC) methodology[25] allows a more semantic characterization of both the nature of the defects found and the thoroughness of the testing process.

Static testing—source code analysis. Analysis of source code to expose potential defects is a well-developed branch of software engineering.[26] The typical mode of operation is to make the target code available to a code analysis tool. The tool will then look for a class of problems and flag them as potential candidates for investigation and fixes. There are clear benefits in source code analysis: it can be done before an executable version of the program exists. A certain class of faults, for example memory leaks, are more easily exposed by analysis than by testing. A fault is more easily localized, since the symptoms tend to be close to the cause. Typical analyses performed will involve a compiler or parser, tied to the language of the program, that builds a representation, such

as a call graph, a control graph, or a data flow graph, of the program. Many commercial and research tools are available in this area.

Test automation. There are four major parts to any testing effort: test case design, test case creation, test case execution, and debugging. Due to the sheer volume of testing required, most organizations are moving from testing primarily in manual mode to automation. Due to the complexity of the issues to be considered, the activity that is most easily amenable for automation is test execution. This assumes that the test cases are already manually defined and written (or captured via a tool) and can be executed in an automated test execution environment in terms of scheduling, logging of results (success or failure), capturing of details of the failing environment, and so on. For testing that requires the explicit use of graphical user interfaces, the automation of test case execution has already produced major productivity gains across the industry. There are a number of commercial and research test tools available.

Automation of test case design (and hence test case creation) is another matter.[11,27] In order to automate functional test case design, we need a formal description of the specifications of the software behavior, resulting in a model of the software behavior. As discussed earlier, this formal description is not captured in typical commercial organizations. While the use of finite state machine technology[28] is beginning to take hold in the model-checking community, its use in the broader industrial software testing community is limited at best. The increasing adoption of UML (Unified Modeling Language) by software developers as the design language may provide the first opportunity for a widely used technique to capture a more formal description of software specifications. However, UML still lacks the constructs to be an effective language for capturing realistic test case specifications.[29,30]

Regression testing. For software that has undergone many releases, one of the nagging questions has been the validation of the specifications of prior releases against the current release.[31] This is typically done via the execution of "regression" test cases (those used to validate prior releases) against the current release. In most real-world environments, there is no automated traceability established among test cases (that is, no one knows why the test case was added or if it is still valid). In addition, the relationship between code requirements and their implementation is not tracked between releases. Hence regression testing not only checks that earlier specifications are still valid, but also catches backward-compatibility problems. While there is a clear need to keep adding to the regression test suite, based on concerns about the cumulative specifications against the current release, the inadequate information captured makes it impossible to prune the regression suite as the product evolves.[32] This is another area where automated test case design can help, since the test cases will be naturally linked to the specification and the traceability will be built in.

Conclusions

We have only scratched the surface of debugging, testing, and verification in this paper. Other significant work can be found on automated debugging,[33] coverage-based testing techniques,[34] performance testing and analysis,[35] and concurrent and distributed testing.[36]

Testing and verification of software, as a discipline, has concentrated on the implementation portion of the software life cycle: take a specification (often written in a fuzzy way) and then write a working program that can be tested or verified. There has been some work on the specification stage of the life cycle: automatically producing formal specifications from the informal high-level descriptions, but there has been little related work on the other end of the life cycle: deployment and maintenance.

Testing and verification are always done in a context and rely on a base set of assumptions. Unfortunately the assumptions are often unstated and are frequently unmet in real-world environments. How often has a new version of a piece of software failed because of some unplanned-for device driver, or some competing piece of software that grabs control of an interrupt? How then is one to test or verify subsequent upgrades of or patches to existing software? There has been work on specifying and testing components and then combining them into full systems, but that work depends on, and thus is no further along than, the general techniques of specification and testing. It falls prey to the same fragility of the assumptions of the individual components. This has been further complicated because the three topics of debugging, testing, and verification are treated by different groups of experts with different backgrounds, different conferences, and different journals, usually precluding any integration of ideas or techniques. This clearly points to the need to treat and teach software engineering as a holistic disci-

pline, rather than a collection of tricks and tools that experienced programmers know how to exploit. Perhaps this special issue can lead the way to this holistic view of the three areas, so that software engineers everywhere can enjoy the fruits of the new technologies that are being developed in the laboratories.

Acknowledgments

We thank Melissa Buco for programming support and Joanne Bennett for administrative support in the planning and coordination of this special issue on testing and verification.

Cited references

1. F. P. Brooks, *The Mythical Man-Month*: *Essays on Software Engineering*, Anniversary Edition, Addison-Wesley Longman, Reading, MA (1995).
2. G. J. Myers, *Software Reliability: Principles and Practices*, John Wiley & Sons, Inc., New York (1976).
3. *Glossary of Software Engineering Terminology*, ANSI/IEEE Standard 729-1983, IEEE Standard, IEEE, NY (1983).
4. M. S. Deutsch, *Software Verification and Validation: Realistic Project Approaches*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).
5. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York (1992).
6. *IEEE Guide to the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Standard 982.2-1988, IEEE, New York (1989).
7. W. S. Humphrey, *Managing the Software Process*, Addison-Wesley Publishing Co., Reading, MA (1990).
8. N. Francez, *Program Verification*, Addison-Wesley Publishing Co., Reading, PA (1992).
9. M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* **15**, No. 3 (1976).
10. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., New York (1976).
11. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York (1990).
12. E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press, Cambridge, MA (2000).
13. *Model Checking Software*, *Proceedings*, *Eighth International SPIN Workshop*, Toronto, Canada (May 19–20, 2001).
14. K. R. Apt and E. R. Olderog, *Verification of Sequential and Concurrent Programs*, Second Edition, Springer-Verlag, Heidelberg (1997).
15. H. Lieberman, "The Debugging Scandal and What to Do About It," *Communications of the ACM* **40**, No. 4, 26–29 (April 1997).
16. M. Eisenstadt, "My Hairiest Bug War Stories," *Communications of the ACM* **40**, No. 4, 31–37 (April 1997).
17. R. Baecker, C. DiGiano, and A. Marcus, "Software Visualization for Debugging," *Communications of the ACM* **40**, No. 4, 44–54 (April 1997), and other papers in the same issue.
18. W. De Pauw and G. Sevitsky, "Visualizing Reference Patterns for Solving Memory Leaks in Java," *Lecture Notes in Computer Science* **1628**, Springer-Verlag, Heidelberg (1999), pp. 116–134 (*Proceedings, European Conference on Object-Oriented Programming*, Lisbon, Portugal).
19. S. Horowitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems* **12**, No. 1, 26–60 (January 1990).
20. W. Polak, "Compiler Specification and Verification," *Lecture Notes in Computer Science* **124**, Springer-Verlag, Heidelberg (1981).
21. E. W. Dijkstra, "Notes on Structured Programming," *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Editors, Academic Press, London (1972), pp. 1–82.
22. D. Hamlet, "Foundations of Software Testing: Dependability Theory," *Software Engineering Notes* **19**, No. 5 (*Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*), 128–139 (1994).
23. J. Musa, *Software Reliability Engineering*, McGraw-Hill, Inc., New York (1998).
24. S. H. Kan, J. Parrish, and D. Manlove, "In-Process Metrics for Software Testing," *IBM Systems Journal* **40**, No. 1, 220–241 (2001).
25. K. Bassin, T. Kratschmer, and P. Santhanam, "Evaluating Software Development Objectively," *IEEE Software* **15**, No. 6, 66–74 (1998).
26. D. Brand, "A Software Falsifier," *Proceedings, Eleventh IEEE International Symposium on Software Reliability Engineering*, San Jose, CA (October 8–11, 2000), pp. 174–185.
27. R. M. Poston, *Automating Specification-Based Software Testing*, IEEE Computer Society Press, Los Alamitos, CA (1996).
28. D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines—A Survey," *Proceedings of the IEEE* **84**, No. 8, 1090–1123 (1996).
29. A. Paradkar, "SALT—An Integrated Environment to Automate Generation of Function Tests for APIs," *Proceedings, Eleventh IEEE International Symposium on Software Reliability Engineering*, San Jose, CA (October 8–11, 2000), pp. 304–316.
30. C. Williams, "Toward a Test-Ready Meta-Model for Use Cases," *Proceedings, Workshop on Practical UML-Based Rigorous Development Methods*, Toronto, Canada (October 1, 2001), pp. 270–287.
31. J. A. Whittaker, "What Is Software Testing? And Why Is It So Hard?" *IEEE Software* **17**, No. 1, 70–79 (January/February 2000).
32. M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi, "Regression Test Selection for Java Software," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Tampa, FL (October 14–18, 2001).
33. A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *Proceedings, 7th European Engineering Conference* held jointly with the *7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Toulouse, France (September 6–10, 1999), pp. 253–267.
34. M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal, "A Study in Coverage-Driven Test Generation," *Proceedings, 36th Design Automation Conference*, New Orleans, LA (June 21–25, 1999), pp. 970–975.
35. F. I. Vokolos and E. J. Weyuker, "Performance Testing of Software Systems," *Proceedings, First ACM SIGSOFT International Workshop on Software and Performance*, Santa Fe, NM (October 12–16, 1998), pp. 80–87.
36. R. H. Carver and K.-C. Tai, "Replay and Testing for Concurrent Programs," *IEEE Software* **8**, No. 2, 66–74 (March 1991).

Accepted for publication November 20, 2001.

Brent Hailpern *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598-0704 (electronic mail: bth@us.ibm.com).* Dr. Hailpern received his B.S. degree, *summa cum laude*, in mathematics from the University of Denver in 1976, and his M.S. and Ph.D. degrees in computer science from Stanford University in 1978 and 1980, respectively. His thesis was titled, "Verifying Concurrent Processes Using Temporal Logic." Dr. Hailpern joined the IBM Thomas J. Watson Research Center as a research staff member in 1980. He worked on and managed various projects relating to issues of concurrency and programming languages. In 1990, Dr. Hailpern joined the Technical Strategy Development staff in IBM Corporate Headquarters, returning to the Research Division in 1991. Since then he has managed IBM Research departments covering operating systems, multimedia servers, Internet technology, and pervasive computing. He was also the client product manager for the IBM NetVista™ education software product, for which he received IBM's Outstanding Innovation Award. Since 1999, he has been the Associate Director of Computer Science for IBM Research. Dr. Hailpern has authored 12 journal publications and 13 United States patents, along with numerous conference papers and book chapters. He is a past secretary of the ACM, a past chair of the ACM Special Interest Group on Programming Languages (SIGPLAN) and a Fellow of the IEEE. He was the chair of the SIGPLAN '91 Conference on Programming Language Design and Implementation and was chair of SIGPLAN's OOPSLA '99 Conference. In 1998, he received SIGPLAN's Distinguished Service Award. He is currently chair of the OOPSLA Conference Steering Committee and an associate editor for ACM's *Transactions on Programming Languages and Systems* (TOPLAS).

Padmanabhan Santhanam *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: pasanth@us.ibm.com).* Dr. Santhanam holds a B.Sc. degree from the University of Madras, India, an M.Sc. from the Indian Institute of Technology, Madras, an M.A. degree from Hunter College, The City University of New York, and a Ph.D. degree in applied physics from Yale University. He joined IBM Research in 1985 and has been with the Center for Software Engineering, which he currently manages, since 1993. He has worked on deploying Orthogonal Defect Classification across IBM software labs and with external customers. His interests include software metrics, structure-based testing algorithms, automation of test generation, and realistic modeling of processes in software development and service. Dr. Santhanam is a member of the ACM and a senior member of the IEEE.