Preface

Customers and independent software vendors have a right to expect high-quality, defect-free products from IBM. The process used for software development has a great deal to do with the quality of the results, and testing is a crucial part of that process. Because the cost of testing and verification can exceed the cost of design and programming, the methodologies, techniques, and tools used for testing are key to efficient development of high-quality software.

In this issue of the *IBM Systems Journal*, an overview essay and nine papers—from IBM, AT&T, and the University of Massachusetts—discuss technology and tools for software testing and verification. We are indebted to P. Santhanam and B. Hailpern of IBM Research for initiating the topic, soliciting papers and mentoring their development, and coordinating this special issue.

Throughout the development process, and especially during testing and verification, potential defects ("bugs") are detected and removed. In the overview essay, Hailpern and Santhanam provide formal definitions for debugging, testing, and verification and show how each activity relates to the software development process. They then discuss recent improvements in technology in all three areas.

Testing can be broadly defined as a form of measurement, and the next three papers discuss aspects of measurement. In the first of these, Bassin, Biyani, and Santhanam describe metrics that can be applied as part of the acceptance test for vendor-developed software. New metrics presented here were developed for evaluating vendor software components to be integrated for use during the 2000 Summer Olympics. The authors explain how metrics can be used to estimate the degree of risk involved in accepting a software component, based on test case execution.

The paper by Butcher, Munro, and Kratschmer describes three case studies in which Orthogonal Defect Classification (ODC) was used. ODC, a quantitative method used to improve processes as products evolve, was applied to a mature product, a large middleware product, and small team project. In each case, the team was able to reach its objective of improving test effectiveness with minimal impact on organizational resources.

Weyuker and Avritzer introduce the "performance nonscalability likelihood" metric, designed to predict the likelihood that the system will fail to meet its performance goals when the workload is scaled. A key step in its application is to identify the key resources that constrain scalability. The authors describe how they applied the metric to a case study of a large industrial production system.

The next two papers discuss the testing process and tools that support it. Loveland et al. report on the process, tools, and techniques used to test z/OS*, the operating system for IBM's zSeries* processors. Many z/OS customers have requirements for continuous availability, and the expectation of "zero down time" places high demands on testing. The authors provide a practical guide to key approaches that have proven effective for testing z/OS and its predecessors.

Because of the variety of platforms and products IBM develops and supports, there are many testing tools with similar functionality within the corporation. Williams et al. describe the architecture, developed by members of IBM's Software Test Community Leaders group, for integrating these tools, as well as new ones. Three integration concerns are addressed: data

across tools and repositories, control across tools, and a single user interface into the tool set.

Part of the difficulty in testing complex software is the very large number of test cases to be designed, created, executed, and debugged. Some of these activities can be automated, as we see in the next three papers. Farchi, Hartman, and Pinter describe their use of a test case generator, based on a finite state machine model derived from natural language software specifications. In two experiments testing for standards conformance, they found that the effort needed to develop a model and run the test cases was less than would be needed for conventional testing, and that the model-based testing gave better code coverage.

Edelstein et al. describe ConTest, a tool that both generates and executes tests for detecting synchronization problems in multithreaded Java** programs. The tool first seeds the program under test with the Java sleep, yield, and priority primitives. As the program is run, the tool decides either randomly, or based on code coverage requirements, whether or not to execute the seeded code.

The paper by Rankin also discusses the automation of test case execution, focusing on the reuse of existing test procedures. The Software Testing Automation Framework (STAF) can be used across platforms and with multiple languages. As an example of its application, the author explains how it was used by an IBM testing organization to automate the execution of a resource-intensive test suite.

In the last paper in this issue, we turn from software testing to software verification. Cobleigh, Clarke, and Osterweil describe a finite state verification (FSV) approach and FLAVERS, its prototype implementation. Given source code and some behavioral properties of the system, FLAVERS attempts to verify that all possible executions of the program will satisfy these properties. Unlike other FSV approaches, FLAVERS generates a compact and conservative model to which precision can be added as required, based on analysis results. Experimental results indicate that the approach can be scaled to address more complex problems.

The next issue of the *Journal* will be devoted to papers on e-commerce and Web services.

Marilyn L. Bates
Associate Editor

John J. Ritsko
Editor-in-Chief

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.