What is testing?

Nearly everyone in the computer business is concerned about quality. It's probably fair to say that many millions of lines of software—the programs that keep the world working—didn't get to that state by any other technical means (such as having been proved correct)! In this sense the task of researchers is to discover what was done and to design methods that accomplish the same (or better) effects at less cost.

Elements of testing technology

The technology of testing computer programs appears to divide into some natural categories, all of which contribute to the basic objective of systematically analyzing the actual behavior of programs:

*Static analysis* seeks to demonstrate the truth of certain allegations about program properties without necessarily having to execute the programs.

*Dynamic analysis* seeks to understand the internal relationships between a program test and the parts of a program that are activated (exercised) during the test.
Test case design attempts to figure out how to construct and/or organize tests to get the best testing effect (highest likelihood of discovering errors) with the least effort.

Symbolic evaluation attempts to determine properties of programs (with a quality level quite close to proof-of-correctness) without actually executing them.

Automated tools provide the technical means to set up, measure, record, and archive the results of testing.

Effectiveness measurement attempts to provide feedback to a user on the quality of past and current testing activities on a particular software system.

Note that all of these subjects relate to a kind of systematic activity that is forward-going in the sense that it assumes reasonably stable programs but expects to find errors. (If there were no errors testing would be unnecessary, but if the programs are too immature even to be compilable there is no dynamic behavior to analyze!) A second point worth emphasizing is that all of these technologies deal directly with the object for which quality statements are ultimately to be made: an actual, “working” (in the sense of executing) computer program. This point is in contrast to other analytic methods that deal in terms of secondary descriptions of programs, either before or after the production phase.

Historical notes

Program testing has, of course, been known ever since programming began. In a manual written in 1950 Turing referred to testing as the extreme form of the experimental method of correctness affirmation.

The main interest in program testing began in the U. S. with the 1972 Chapel Hill Conference on Computer Program Test Methods and the resulting book by Hetzel. A measure of the growth of the field can be had by noting that Hetzel’s book had a bibliogra phy with about 200 citations relating directly to testing, whereas the text distributed with the IEEE Computer Society’s tutorial on program testing given at COMPSAC 77 has over 400 citations on the subject.

Another landmark of program testing technology is the paper by Goodenough and Gerhart, the first serious attempt to put program testing on a sound theoretical basis. That paper fostered interest in testing methods that have lead to some surprisingly strong theoretical results. The more recent work of Howden is of particular importance in this area.

Where is testing today?

The five papers in this issue represent an excellent cross-section of program testing technology as it is known and practiced today. Each of the papers deserves special comment.

- Static analysis and dynamic testing. Richard Fairley introduces the notion of software validation as encompassing a range of static and dynamic methods for demonstrating the correspondence of a program with its specifications, taken as the “standard” of comparison. The static analysis process is supported by theoretical results and methods arising from program optimization theory involving control-flow and data-flow graphs. Fairley discusses bottom-up, top-down, and mixed system testing strategies and suggests how these can be intermixed with automated dynamic testing.

Program instrumentation. Program instrumentation is a way of learning about the effect individual tests have on a program. It has some attractive advantages in terms of economy and effectiveness. Even though the instrumented program’s execution time and space are affected, the process is so simple and straightforward that it can be automated quickly and easily. The result is a simple-to-interpres measure of testing coverage.

Moreover, as J. C. Huang points out in his tutorial paper, it is possible to provide instrumentation that gives a user information about the internal values computed during the program’s operation. In addition, it is possible to detect certain kinds of data flow anomalies entirely through program instrumentation.

Test data selection. DeMillo, Lipton, and Sayward introduce a new, empirically observed effect, which they call “the coupling effect” that may become a very important principle in practical testing activities. The idea is that programs appear to have the property—the “coupling effect”—that tests designed to detect simple kinds of errors are also effective in detecting much more complicated errors. This relationship may seem counter-intuitive, but the authors give a way of analyzing it through the use of program mutations (i.e., incorrect variations from a correct program). One of the most interesting possibilities is that the mutation idea could form the basis for statistically inferring the likelihood of remaining errors in a program.

Automatic software test drivers. David Panzl addresses the all-important automated-tools area. Program testing, if it is to be comprehensive, generally must be accomplished with the assistance of automated tools simply because there are too many steps, and too much resulting data, to permit a good job to be done by hand. Panzl discusses a new form of tool called a test driver. The test driver automates the process of (i) planning a specific test of a program, (ii) setting up and running that test and collecting the results, and (iii) analyzing whether the results produced are correct or not. The tool eliminates the need for manually-generated stubs or special test-drivers.
Applications of symbolic execution to program testing. While the prior papers treat testing in terms of running actual programs with actual test data, Darringer and King abstract that process and discuss the possibilities of purely symbolic execution. Their symbolic execution tool operates by generating symbolic execution trees that are analyzed interactively by a user. The important point about this tool is the possibility it offers for bridging the gap between a good set of tests and a formal proof of correctness. Of all the developments in program testing technology in recent years, this one is the most exciting for workers in the field.

Acknowledgments

Besides the authors of the papers, a large number of people contributed their time and energy in one or more of the following ways: by contributing a paper that was considered but ruled out because of its nature (all such papers are being published elsewhere), by refereeing a paper, by suggesting authors and/or referees, or by advising the guest editor. All of these individuals deserve a special vote of thanks:

Dennis L. Baggi
Polytechnic Institute of Brooklyn
G. R. Baldwin
Standard Oil of California
Allen E. Beutel
Naval Ocean Systems Center
John Bielski
Honeywell, Inc.
George R. Cannon
Logicon, Inc.
John D. Capps
NASA/Huntsville
W. C. Carter
IBM Research
Lloyd Fosdick
University of Colorado
John Goodenough
Softech, Inc.
Mary Ann Goodwin
NASA/Houston
Jack Grimes
Tektronix, Inc.
Richard Hamlet
University of Maryland
Herb Hecht
The Aerospace Corporation
Bill Hetzel
Blue Cross/Blue Shield of Indiana
Richard Kleir
Data General Corporation
S. Kundu
Logicon, Inc.
Donna Leland
Software Research Associates
M. Lipow
TRW
Thomas McCabe
National Security Agency

Eldred C. Nelson
TRW, Inc.
George Petteys
TRW, Inc.
Vic Schneider
The Aerospace Corporation
N. F. Schneidewind
Naval Postgraduate School
Martin L. Shooman
Polytechnic Institute of Brooklyn
Leon Stucki
Boeing Computer Services
Richard Thayer
U.S. Air Force
Irv K. Wendel
SWAK, Inc.

References