The field of software testing spans mathematical theory, the art and practice of validation, and methodology of software development. To cover this range would require a textbook (or several texts), not a trio of articles. But the work presented in this special section is a kind of "test set." Each paper is a significant contribution within one of the three broad areas. The reader must now make the assessment that is critical to any review of test points: are they representative? My own answer is 'no'; these articles are provocative and revealing rather than routine summaries. And perhaps that is what software testing is all about: good tests are the ones that provide new insights, not the ones that cover well worn ground.

The Evaluation of Program-based Software Test Data Adequacy Criteria

The twin foundations of any scientific or engineering discipline are experimental knowledge and abstract conceptualization. Progress rests not only on "knowing what works," but on a framework for organizing and thinking about what is known. Without a theoretical framework, even the best experimental results cannot be fitted into a useful whole. Author Elaine Weyuker has long advocated attention to the foundations of program testing, and has used her background in mathematical logic and recursive function theory to contribute to those foundations. Her article is a study of the logical basis for structural testing. By critically examining the properties that program-based testing methods must possess, she seeks to characterize those methods in a mathematical theory.

Perhaps the most interesting part of this work is its analysis of conflicts between "obvious" properties of testing methods. For example, Weyuker has shown that there is a logical conflict between insisting that tests succeed and calling a more inclusive test better [22]. Weyuker's theory is in its early stages: the axioms she has in hand do not yet characterize test adequacy. In her article she analyzes examples that "escape" characterization, and seeks ways to close these loopholes.

The Category-Partition Method for Specifying and Generating Functional Tests

The paradox of program testing is that its techniques are compute-intensive, but there is a deep-seated resistance to computer aids. Research tools, many of them originating in practical environments, are well understood but woefully little used [24]. The only explanation seems to be that testers are too short on resources to consider automating their tasks, reasoning that is acknowledged to be short-sighted, but which too often carries the day. Thomas Ostrand and Marc Balcer have written a compelling counterexample to show how helpful computer assistance can be at modest cost. They describe a test-description language and a processing tool that isolates functional test cases, with the aim of providing specification coverage in a minimum number of tests.

The Growth of Software Testing

David Gelperin and Bill Hetzel begin their article with a history of program testing; no one is more qualified to characterize the field. They show how debugging and testing diverged in their aims, how demonstrating that programs work gave way to searching for program faults, and how detection methods are now leading to...
fault prevention. They argue that "testing" has a wider scope in program development than merely the execution of programs. The testing viewpoint can be invaluable in providing feedback to improve specifications and designs as well as code. Early in a program's development, testing feedback leads to improvements so that potential errors never appear in the final code.

AN ESSAY ON TESTING THEORY AND PRACTICE
My interest in program testing springs from practical and theoretical roots. While I was a student attempting to characterize properties of programs that could be pinned down by testing [7], I was also a systems programmer responsible for maintenance and reliability of two large operating systems [8]. I was introduced to computer science by Halstead's wonderful book on NELIAC self-compiling compilers [6]. What fascinated me was the prospect of capturing an infinite object in a finite description. (I thought, at first, that a self-compiling compiler did just that: in one program it held the seeds of all programs.)

Program testing has exactly the right character. Programs are complex, hard to understand, hard to prove, and consequently often riddled with errors. But might not a small set of tests, conducted with ease, pinpoint problems for repair? And when all problems have been corrected, is the program not perfect? Testing seems far easier than programming, certainly easier than proving a program. The infinite complications of what the program does, what it is supposed to do, etc., are replaced by a finite set of data. Programs are complex, hard to understand, hard to prove, and consequently often riddled with errors. But might not a small set of tests, conducted with ease, pinpoint problems for repair?

In principle, it could work that way. The defect rate for carefully made programs can be about 5 faults/1,000 lines, so that a 1M-line program might have only 5,000 problems. Even supposing that a unique test point is needed to expose each one, thousands of trials are not impractical. The fallacy may be obvious, but it underlies our intuition about tests more than we like to admit: the points may be few, but we do not know how to find them. Worse, we know no systematic way to search, no way to judge points selected, and no way to decide when to stop. So when 6,000 points have been tried and no errors found, we do not know if there are none, or if 10 remain, or 5,000.

Formal testing research is only about 15 years old, and its results are modest. But even what is known is not finding its way into practice very well. This essay explores testing theory and practice, suggests reasons for the present situation, and predicts future work.

TESTING AND SOFTWARE ENGINEERING
I would describe software engineering as heavy-handed management based on modest common-sense principles. I do not intend to disparage the work on structured programming and development methodology, which has made large-system development a predictable craft instead of an arcane art. But its technical basis is not impressive. Modular, top-down programs were written before 1960 [15], and recognized as a good thing. To enforce such standards across the board is a management breakthrough, but not a technical one. The Gelperin and Hetzel article would argue that a great deal can be done in testing methodology, and that the proper arena is not the narrow one of programs and their execution, but rather the whole scope of development. But it is the technical problems of software that interest me: what can we do automatically, that necessarily amplifies the intelligence and effectiveness of programmers, that makes their work more rewarding and less routine?

The field of programming languages provides a technical model. Languages and their compilers have theoretical backing and tremendous practical utility. Indeed, high-level languages enabled most of the software-engineering common sense in use today. Unless program testing has a similar technical base, software-engineering methodology for testing is premature. Programs must be tested, however little we understand the process, and managers must try to find out what works, but strict legislation based on wishful thinking is a prescription for disaster.

The technical discipline of testing can be seen as theory versus practice. Practical testing began with the first program, and so is at least twice as old as formal research in the subject. Clever programmers, often without understanding why things seem to work, have devised many testing methods, some with great intuitive appeal. In trying to analyze these methods, testing theoreticians have only been able to show that in the worst case, all fail [12, 23]. Perhaps by accident, theoreticians have come from absolute disciplines like logic, program proving, and algebra, and they have sought absolute results. They wanted to discover how testing is related to correctness. In general the two are unrelated, so the theoretical results are negative. Whatever we would like to do is usually an unsolvable problem. When we isolate tractable special cases to dodge the unsolvability, they turn out to be too narrow and atypical.

SOME TYPICAL RESULTS
I will describe the state of the program-testing discipline through an example. The method and results are not the most recently proposed, but I think they are typical of the field. Branch testing is a method that arose in practice, as a refinement of trying to execute every statement of a program. Every conditional expression in a program is singled out. A collection of test points achieves branch coverage just in case each conditional is forced to take both its true and false branches by the
test collection. The special case of an IF-statement with no ELSE in which the condition fails is of particular interest.

Branch testing, like most testing methods, is based on coverage, in this case of the alternatives of conditional expressions. The idea is that if something is not covered, then that something might hide an arbitrarily bad blunder. A secondary virtue of coverage is that seeking it is systematic—the person doing the testing has a narrow, directing goal. The flaw in branch testing is that bugs on some branch may not be excited unless some particular control sequence led into the branch; or, unless some particular internal values were assumed when the branch was taken. (Attempting to plug those holes leads to variants of path testing [11, 14] and mutation testing [4, 9, 13] respectively; these in turn have their holes—there is no end to it.) It does not take theory to expose the failings of methods like branch testing—real bugs in real software do it soon enough. The bugs emerge when a customer’s data takes a branch under circumstances that were not tested.

Branch testing is very easy to monitor. The typical implementation uses a preprocessor to insert statements into a source program, statements that record when control has reached each branch point. The data structures updated by these statements can be analyzed at the end of all test executions to discover missed branches. Such implementations have been invented over and over for different languages and environments since the 1960s. One with good graphics support can be helpful at low cost, but almost no commercial program developers routinely use such tools.

Can tests that achieve branch coverage be automatically generated? This question verges on testing research rather than techniques that many programmers reinvent, and the answer is a typical theoretical one.

Theorem: The problem of deciding if an arbitrary branch in an arbitrary program can be taken at all is undecidable.

(The easiest proof is a “reduction of the halting problem,” which works by showing that if one could identify impossible branches, one could discover termination of programs, known to be impossible. The essential idea is to put branches at a program’s HALT statements.)

In practice, reasonable special cases exist that vitiate this result. If all conditional expressions are linear (expressed in terms of input values), then the conditions describing entry to each branch are sets of linear equations, whose solution is straightforward and mechanical. Solutions to the equations are data points that will force branch coverage [2, 17]. Investigation of such special cases is typical of testing research of the mid-1970s. Again, a useful tool can be constructed using these results, and research versions have long been available. Despite low cost such tools are very infrequently used.

The theory and practice of branch testing are typical of existing methods. Each technique has around it a similar web of results, some very general (and usually negative), some practical, some embodied in usable tools (which nonetheless seem to be little used). It is easy to lose sight of the fundamental deficiency in the results: branch testing that finds no errors has no significance for the quality of the tested program. If a branch has never been tried, then certainly a bug there will not be found. But the real questions are the reverse ones: if a program has bugs, will branch testing necessarily find them? If branch testing finds nothing, is there nothing to find? The answer to both questions is: “not necessarily.” The method is systematic and can be automated, but its value is unknown. Perhaps that accounts for its lack of acceptance.

Thus, testing theory has failed in its analysis of particular methods. It has also failed to yield any meaningful comparisons of methods. The only available comparison is the “subsumes” ordering [18]: method A subsumes method B just in case any test meeting the A criterion necessarily meets the B criterion. That is, having conducted a test according to method A, there is no need to conduct one according to method B, because its criterion has necessarily been met. Unfortunately, most even slightly different methods are not comparable—neither subsumes the other [21].

EXPERIMENTAL EVALUATIONS

It seems obvious that experiments should be tried to learn the efficacy of testing methods that cannot be absolutely justified [1]. What is unsolvable in the general case may have an efficient algorithm for a special case, and just so the “not necessarily useful” of the general case might turn into “very useful” for (say) branch testing of real programs. However, experiments face two formidable problems of standardization.

The lesser difficulty is one of classification: what exactly is a “real” program? The program size, the size and experience of the development group, the resources used and the time scale, and most of all the nature of the problem to be solved, are certainly factors, but how will they be quantified and controlled? Without good characterizations, results of one experiment may not apply to another situation. We do not know exactly what is important, and shotgun experiments in which everything is combined are not revealing. Attempts to isolate factors have also failed, because with so little understanding, it is impossible to know if other unknowns have been held constant with values that do not distort the results.

The more serious difficulty is distortion of testing experiments caused by human skill and chance. The human subjects of an experiment pick the test points. They are seeking (say) branch coverage, but there are many ways to attain it, and a person’s subjective choice influences the result. I call this the “nose-rubbing” effect, after a method of house-training puppies in which accidents are called to the puppy’s attention by rubbing its nose in the mess. The name is appropriate not only because buggy code is a mess, but because the puppy is
expected to do some thinking: we are not calling for more of the same in the same place.

Suppose a programmer attempts to attain branch coverage, and all test points produce correct results, but some branch remains unexecuted. In the process of finding data to cover it, a fault is found. This can happen because:

1) The fault is seen while studying the program to find additional test points.
2) Additional test points are tried that fail to cover the branch, but happen to produce incorrect results.
3) Additional test points do cover the branch, and also produce incorrect results.

The experimental conclusion that “branch testing found the fault” is questionable. Perhaps in case 1 the study was in a different part of the program from the untried branch, and the problem had nothing to do with that branch. In case 2 the branch seems irrelevant. Case 3 seems to support the conclusion, but suppose that the incorrect results are traced to logic that should have been inserted before the branch? In such circumstances it is only accidental that the error was found while doing branch testing. The person found the fault, and maybe the “method” did no more than keep the person interested in more tests and more study. If branch testing can be credited with finding the fault, so could any method that rubs the programmer’s nose in the code. (For example, upon being presented with any test results at all, a manager could say, “Not enough! Try to exercise routine (fill in the blank) more!”) Although no study has been attempted to demonstrate the distortions of nose-rubbing, evidence for it exists. In [1] it was found that a coverage technique detected about 40 percent of the faults of omission in code (and another 20 percent could have been seen but were simply not noticed). It is ludicrous to say that covering something as that for the program’s long term operation. On this basis it is possible to model the testing process, for example to compute the number of successful test points needed to guarantee that R is below a given value, with a given confidence [5]. Or, the mean time to failure for a continuously operating system can be predicted from measurements of the time sequence of its crashes [20].

Programming skill (and presumably testing skill is related) is one of the most widely varying aspects of human behavior. Therefore, testing experiments suffer maximum distortion from the nose-rubbing effect. Comparisons between methods are doubly uncertain, because the skills needed to use the methods may be different. This points out another defect in the subsumes ordering: an experiment can show that method A is inferior to method B, even though A subsumes B. The technically more powerful method could be harder to use, and so people choose less useful data to satisfy it. The fault lies both with the subsumes ordering and with such experiments.

Management based on this shifting ground can succeed, but relies heavily on the goodwill of technical staff. Ordered to conduct (say) branch testing, which is unreliable in itself, the staff can fulfill the letter of the law without real effect. However, branch testing may be as good any other nose-rubbing technique if used creatively with intent to discover problems. It is almost as if a hardline manager relied on a time clock to check employees’ time sheets, but the clock does not keep time, so the people have to set it before punching in and out. Programmers have a real interest in making their code work, and there is an immense store of goodwill on which management can draw, but it is ultimately dangerous to put trust in unsound technical procedures. Under pressure of tight schedules, or because of personality conflicts, cheating might become desirable.

Programming skill (and presumably testing skill is related) is one of the most widely varying aspects of human behavior.

STATISTICAL THEORY

Had the dominant theoretical view of testing been probabilistic, the field might look quite different. There is nothing very clever in random sampling of program inputs, and the result cannot be expected to magically show correctness, but statistical measures seem just the right way to evaluate the dubious experiments that tests are. Intuitively, one of the reasons we trust programs is because they have a long history of working properly, and statistical quantification of this idea should be investigated.

Most of the statistical work is based on the failure rate model [20], in which it is assumed that a program has a probability R of failing for a randomly selected input, and thus that as more and more tests are conducted, the observed number of failures will approach fraction R of the trials. It must further be assumed that random selection during the tests follows the same distribution as that for the program’s long term operation. On this basis it is possible to model the testing process, for example to compute the number of successful test points needed to guarantee that R is below a given value, with a given confidence [5]. Or, the mean time to failure for a continuously operating system can be predicted from measurements of the time sequence of its crashes [20].

Or, the total time required to debug a program can be predicted from its initial history of problems and fixes [19].

The failure rate model is not widely accepted. Its assumptions do not seem appropriate for programs, and the mathematics is tractable only for the simplest cases. Some of the results are intuitively wrong. For example, the number of tests required to force high confidence in a low failure rate does not depend on the size of the program or on the size of the input domain. Critics explain successful empirical studies (particularly for models of the debugging process) as curve fitting with an adequate supply of parameters. However, even if failure-rate models are wrong, probabilistic analysis seems appropriate for testing theory because it is capa-
FUTURE NEEDS AND DIRECTIONS

From the technical side, what testing needs most is a sound fundamental theory, one capable of analyzing and comparing particular methods. The “absolute” approach has shown that in general nothing but exhaustive testing can be relied upon, and that special cases do not escape this result except in unreal situations. One promising direction seems to remain, so-called “error-based” testing [16]. Its roots are in mutation theory and hardware testing for specific defects. The tester begins with a precise list of code faults that are to be precluded, and devises tests that can succeed only if those faults are not present. Running the tests then shows not that the software is correct, but that it at least does not contain the faults originally listed. Error-based testing theory requires elements of program proving, in the analysis that demonstrates test points would expose the given faults, and it is a difficult field in which work is only beginning.

There are two candidates for a general theory of testing:

1) Statistical methods can be used for precise analysis, but they are limited to measures such as the number of test points needed. Although statistical theories are routinely used for other kinds of quality control, their application to program testing is suspect because it relies on the failure-rate model. There has been very little progress in finding better theories, perhaps because researchers with appropriate training in probability lack an intuitive grasp of testing issues, and those who understand testing know nothing of decision theory.

2) An axiomatic approach to program testing has broad promise because it can be applied to any property of tests or test methods. A collection of axioms attempts to formally capture properties that tests must possess. Each axiom is self-evident, but their interplay can lead to deeper results. Weyuker’s article represents the beginnings of axiomatics.

Technology transfer from testing research to industrial software development has been slow, and the methods and tools that have been devised have not been widely accepted. This seems strange, because testing tools are often easier to build than those for the now ubiquitous high-level programming languages. Perhaps the reason that testing technology goes unused is that languages must be acquired first and can stand alone, while test systems rely on language details and the operating environment, and have low budget priority. Standardization of operating systems and languages, which has accelerated in both the PC and microprocessor worlds, may make a difference in the near future. The article by Ostrand and Balcer in this section is a good example of what should be happening far more frequently. Their testing methodology is adapted to a special-case problem, and supported by an easily implemented tool.

Decrying the lack of testing foundations while advocating the spread of the technology based on that theory may sound like talking out of both sides of the mouth. But well engineered test tools are worthwhile, even when they use something like branch testing that cannot be relied on to find errors. Nose rubbing works: programmers who systematically look for bugs find and fix them. A tool that checks branch coverage costs very little to build and use. Other tools (such as symbolic execution systems [2, 3]) require skill to use, but are powerful in talented hands. Debugging tools are the best, because they give nose-rubbing full play, and an error found is a gain that in no way relies on a missing theory. Such tools are best placed in the hands of programmers who have the greatest knowledge of and interest in their code. Confidence measures based on testing are not as good, because they are not credible without a theory to back them up, and because they are used by evaluation or management people who do not know the code.

The raw material of sound theory is experimentation. Using a poor debugger shows what it lacks; using a false model for reliability can expose its flaws and suggest changes.

Another reason for advocating the routine use of imperfect test methods is the need for empirical information gathering. The raw material of sound theory is experimentation. Using a poor debugger shows what it lacks; using a false model for reliability can expose its flaws and suggest changes. The infrastructure needed to put testing methods in place is slowly built, and built relatively independent of any flaws in those methods. The danger, of course, is that a premature attempt to use the technology will give it a bad name without collecting information for improvement.

New languages and new styles of programming require new intuitions about testing. The whole of so-called structural test methods (e.g., branch testing) is based on the imperative, state-dependent execution model of von Neumann machine languages and unit testing in conventional high-level languages like FORTRAN and C that make efficient use of such machines. Functional and logic-programming languages do not share these properties. Conventional parallel programs have the usual properties, but repeated in ensembles that add a new dimension of complexity. The methods and intuitions built up over 30 years of conventional program testing do not apply in these situations, which are growing in importance. The opportunity exists to do better testing theory for these new programs than we have done in the past.
Acknowledgments. I have profited from discussions with the authors of the papers included in this special section. While their work passed through a refereeing process, mind did not, so I bear full responsibility for this opinionated essay.

Richard Hamlet
Guest Editor
Department of Computer Science and Engineering
Oregon Graduate Center
Beaverton, OR 97006

REFERENCES

COMPUTING TRENDS IN THE 1990'S

1989 ACM Computer Science Conference.

February 21-23, 1989
Commonwealth Convention Center
Louisville, Kentucky

acm.