Workshop Report:

The workshop's general and session chairmen offer their summaries of the challenges to the software testing community identified at the December meeting.

Software Testing and Test Documentation

Program chairman's overview

Edward Miller
Software Research Associates

The technology of software testing has lagged behind that of other sub-disciplines in software engineering, possibly for lack of interest and possibly because software testing problems tend to be subtler and (some say) harder than thought at first glance. This workshop, the first of its kind, brought together some 60 technical experts from all over the world at the Bahia Mar Hotel in Fort Lauderdale, Florida, for an intensive two-and-a-half-day series of meetings and discussions.

One of the workshop's goals was to identify, if possible, some research and development objectives for the next decade of work on software testing technology. Whether that was accomplished is unclear, but there is little doubt that some interesting questions were raised regarding testing, its relation to other sub-disciplines in software engineering, and its application in general. This report attempts to outline some of the directions the technical presentations took. The articles following this overview take a more detailed look at the individual papers presented.

Theoretical aspects. This session, chaired by Susan Gerhart of the Information Sciences Institute, addressed the underlying foundations of testing. What is the relationship between testing and other methods of quality assurance, most importantly program proving? Under what conditions can a test of a program guarantee correctness, and what are the limits of software testing? The session centered around such questions and the series of discussions they elicited.

Not surprisingly, the session papers focused on Goodenough and Gerhart's excellent technical contribution, which has acted to establish the tone and framework for investigating testing technology over the last few years. Papers by Hamlet, Richardson, and Ostrand and Weyuker addressed these and other issues.

Empirical studies of effectiveness. While software testing theory represents one approach to developing a firm basis for the technology, another involves empirical studies of the effectiveness of various testing methods. William Howden, associate professor of mathematics at the University of Victoria, led this session centering on establishing a good method for finding program errors. There was almost universal agreement that conventional path-based testing methods have some deficiencies, but also that they offer many practical advantages. One presentation dealt with real-world experiences with path testing, another with formal procedures for accepting software, and a third with experiences in a large-scale testing "factory."

Test documentation. Regardless of the testing method used, the tests have to be documented. This session, chaired by workshop co-chairman Herbert Hecht, SoHaR, Inc., addressed procedures for documenting program test data, test coverages, etc. One presentation focused on test design for highly modular programs; another dealt with comparisons of alternative test data collection methods. Perhaps as important as the issues discussed was the formation of a new working group on software testing documentation standards.

Test tools. B. Chandrasekaran chaired the session on alternative test support tools. Tools are important to testing because we must usually process far more program text than we can reasonably handle by any other means. SADAT, a highly automated tool used at a German research in-
Another new approach involves the use of domain analysis on the input space of the program to be tested. The concluding presentation discussed the possibilities of full-proof-level sophistication.

The workshop pointed out a number of facts about the current state-of-the-art in software testing—that the "art" is in fairly good shape but that the "science" needs much more attention; that there is a growing body of experience but that more of it needs to be communicated outside the small (but active) community of researchers; and that there is a growing interest in developing, at low cost and with high effectiveness, practical strategies applicable to contemporary software systems.

Acknowledgments

The support and encouragement of Herb Hecht and Len Birns, the workshop co-chairmen, is gratefully acknowledged—without their excellent work it never would have happened! Dennis Fife handled much of the logistical work beautifully, and we all have him to thank for the NBS support. All the session chairmen deserve special thanks for putting up with weird deadlines and impossible demands—all of which they accomplished with speed and care. Finally, Alta Swander, of Software Research Associates in San Francisco, deserves all our thanks for handling the details so very well.

Reference


Testing theory: field needs definition, experimentation

Susan Gerhart
USC Information Sciences Institute

The theory of program testing is not well developed, although there are results and research activity that did not exist five years ago. One possible reason for this lag in development is simply the failure to realize its necessity. Testing tools are attractive, but the usual complexity of engineering them and the subsequent difficulty of evaluating their effectiveness can both be traced to the lack of a theoretical framework. And until recently another reason has been neglect of the area by academic researchers. They have found program proving far more attractive, with its logical mathematical origins and possible integration with the programming process, than testing with its statistical and experimental origins and a posteriori programming phase. A final reason is simply that the testing problem is hard—researchers have obviously been faced with unsolvability and, until recently, have lacked even the basic definitions to get started.

It is not clear what a theory of testing should look like. Certainly there is a mathematical—either logical or statistical—kind of theory. But testing is also a more general problem related to experimental methodology—how does one determine that the results of an experiment (perhaps a test of a program) are not biased, either systematically or randomly? Testing is also an aspect of management theory—how does one determine the quality required of a product as well as the quality of the product actually produced? Psychology even plays a role—what errors do people make and how do they detect them?

The first paper presented in the session on testing theory, by Weyuker and Ostrander, examined the basic definitions (translating experimentation terms into logic terms) of Goodenough and Gerhart. They proposed a test data selection property called "revealing," which emphasizes concentration on specific types of errors for a subdomain of the program. Richardson's paper developed a set of definitions for discussing symbolic execution of programs and for handling consistency and discrepancy with respect to specifications. Hamlet focused on that property of a test called reliability—its ability to show errors as opposed to those that would be shown by some other test. He examined various restrictions of programs, specifications, and test criteria that would lead to reliable tests. In other sessions White and Cohen developed a testing strategy called "domain testing" and a methodology for studying its properties. Budd followed the new approach of program mutation,
which proves that with respect to some larger class of programs either a given program in the class is correct or no program in the class is correct. He investigated several classes of Lisp programs, showing that testing is sufficient to determine which of the two situations holds for a given program.

Other papers at the conference dealt with subjects directly related to testing theory. The development of a solid theory must proceed in conjunction with experimentation, both in the design of experiments on effectiveness of various testing strategies and in the explanation of the results of such experiments. Even just the description of testing tools requires some basic terminology, e.g. "path," that a theory should make precise.

Advances in the theory of testing will require (1) more and better definitions, (2) more substantial results (theorems) about the feasibility and effectiveness of various testing strategies, (3) the integration of programming methodologies, such as abstraction and structuring, into the assumptions about programs being studied, and (4) more highly insightful experiments, such as those of Howden, which combine the mutual drives of empirical and theoretical studies.

Empirical studies: case studies, experimentation, comparisons offer answers

William E. Howden
University of Victoria, Canada

Several studies of formal theoretical methods in testing1,2 indicate that it may be possible to prove theorems about when a testing method is guaranteed to find errors, but at present we must rely primarily on empirical studies of the effectiveness of different techniques.

Empirical studies can be classified into three broad categories. The first category includes case studies describing the effects of using a method in a particular software development project. The drawback to case studies is the difficulty in comparing the approach used in one project with another approach in a totally different project. Since it is impossible to conduct experiments in which large expensive development projects are repeated several times using different methods, we will have to continue to rely on case studies for much of our important data.

The second category involves controlled experiments in which different teams use different methods for the same set of programs. We can construct an experiment in which we can determine the statistical significance of differences in the effectiveness of the methods.3,4 This approach, however, cannot be easily used to analyze the effectiveness of methods for large systems.

The third category includes studies in which the properties of particular methods are analyzed over carefully selected sets of examples. Studies of this type are not case studies, since they are not "incidental" to the development of a real piece of software. They are not statistical experiments and, in general, involve well-defined methods whose properties can be studied without the need for such statistical experiments. The study of the effectiveness of different testing methods described in Howden5 is of this type.

Session topics. The empirical studies session included the presentation of three papers. Sorkowitz and Budd and Majoros5,6 presented case studies which described the use of a particular collection of techniques. Budd and Majoros offered a number of important insights into the problems of staffing, funding, and operating an independent software testing facility. Sorkowitz described the use of an independent quality control (i.e. validation) team. In this case the team is part of the same facility as the development team but is responsible for certification of software.

The remaining paper, by Woodward, Hedley, and Hennel7 described the results of an analysis of a set of "path coverage" testing techniques. The paper contains an interesting set of figures on the number of unfeasible (i.e. unexecutable) paths which are found in programs.

Several important trends and results were identified in the session's papers and discussions. One was the indication that independent quality assurance groups can function effectively and are an important part of the software development process. The importance of measuring the coverage of a set of tests was re-emphasized. Both simple branch-execution statistics or more sophisticated methods like those proposed by Woodward et al can be used. The surprisingly large percentage of paths that were found to be unfeasible by Woodward indicates the importance of carrying out careful analyses of the advantages and problems of particular methods.

Workshop themes. One of the themes that came up repeatedly during both the empirical studies session and other sessions was the need for research projects analyzing the effectiveness of particular testing methods over realistic sets of programs. The workshop emphasized that an important goal of both theoretical and empirical studies should be the construction of guidelines a programmer or manager can use during the validation phase. It is not enough to know that a particular method finds "a substantial number of errors." We must have answers to questions such as "how does method A compare with B over programs of type C?" and "why is A better than B over programs of type D or errors of type E?" The answers to these questions must be either theoretically provable or built on solid empirical evidence. The workshop encouraged future research of this type as well as continued careful analyses of case studies.

References

Software test documentation: standards will help

Herbert Hecht
SoHaR, Inc., Los Angeles

In all practical software undertakings there is sufficient diversity of interest between developer, tester, and user to require some formality in the communication between them. Documentation is therefore an essential part of the software test activity, regardless of test methodology or the extent of test. In applications software affecting public safety (e.g., aircraft flight control or programs dealing with the operation of nuclear reactors) test documentation is required to satisfy regulatory agencies’ concern with the completeness of testing. The documents become the primary means by which an agency can judge the adequacy of the software product for critical applications.

For military software, the test sequence and to some extent the test documentation are governed by MIL-STD-483, but in general there are few guidelines for either content or format of software test documentation. In practice it ranges from a simple statement that a test was conducted on a specified program with satisfactory results, to furnishing multiple file boxes full of program listings, test stimuli, and test output. Neither of these extremes serves the typical need of a using, supervisory, or regulatory organization for information on the conduct and results of a test. The lack of formal and informal standards for software test documentation, together with the generally small concern for documentation when contracts are written, can lead to unpleasant surprises and much acrimony at the end of a project.

In this connection it is gratifying to report that workshop attendees formed a study to work on software test documentation standards. The effort is headed by Leonard Birns and will function under the Computer Society Technical Committee on Software Engineering as a task group of the Subcommittee on Software Engineering Standards. For those interested, Dr. Birns may be contacted at Computer Sciences Corporation, Route 38, Mooresstown, New Jersey 08057. All members of the IEEE Computer Society are eligible to join the Task Group on Software Test Documentation Standards.

The papers presented at the documentation session dealt extensively with test planning and organization. “A Software Testing Design Technique for Modular Programs,” by F. Woodall and D. Dancesia (both of Booz-Allen Applied Research) and D. L. Cooper (of the Air Force Armament Laboratory at Eglin AFB), emphasized the integration of testing into the design and development process. The software procurement was structured into several “builds” staggered in time. This not only provided minimal capability to the customer early in the program but also leveled the workload and optimized the use of skilled test personnel. As a result testing required only about 15 percent of the total project resources, an astonishingly small fraction. Test documentation relied to a great extent on other documents generated during development and generated as part of configuration management.

“Software Reliability Testing—Has the Best Approach Been Found?” by John B. Bowen of Hughes-Fullerton examined current software reliability metrics and their effect on documentation. While professional publications emphasize normalized metrics (errors per 1000 statements or failures per unit run time), the military requirements set specific thresholds, typically numbers of failures of several severity levels allowed during a demonstration. The question posed in the paper’s title is answered largely in the negative because there is not enough knowledge to identify both development and application factors affecting software reliability.

The session discussion explored the fact that software testing and test documentation comprise much more than program testing. Thus test documentation must already have been generated when software requirements are verified against system requirements, or when the software specification is verified against the requirements. For a typical medium or large software development at least the following test documents must be generated:

- Test procedure—a complete listing of the test conditions, the test environment, and the expected results for each phase of test;
- Test report—containing the test results together with explanations of observed deficiencies, changes made during testing, and a discussion of the relevance of test results against software and system requirements.

To permit user access to the information thus generated, the reports must be structured in a hierarchical manner, with the top level addressing only those aspects significant for system deployment. Subsequent levels (with references where appropriate) will contain additional details. Satisfaction of development requirements and of specific test requirements should be documented at the lower levels. Each level, however, should contain at least the following information:

- Definition of the software under test;
- Definition of the test environment;
- Description of test criteria;
- Discussion of the completeness of testing;
- Test results, including identification of failures due to the software under test and those due to the test environment; and
- Identification of all changes made during and as a result of the test.

A standard format will make contracting easier for both the user and the generator of test documentation, and of course will greatly enhance the understanding of the test results in subsequent reviews. Because good criteria for test procedures, tools, and even results are not yet available, one would also like to see presented in the final documentation data generated during testing that can shed light on these criteria (even when the data are not specifically required by the user). Among metric quantities of interest here are error density and failure rates (for each severity level), as well as the non-metrics relating to error types, test tools employed, and development and test methodology. With some information of this type, future specifying and designers of software will have an objective body of knowledge at their disposal.
Test tools: usefulness must extend to everyday programming environment

B. Chandrasekaran
Ohio State University

Current software testing tools are designed to help the testing process by highlighting different sources of error in different ways. Each tool implements a particular theory or approach believed best suited for uncovering certain types or patterns of errors. Increasing evidence indicates that no single tool will suffice to test a complex program. A well-integrated collection of tools with each tool appropriate for certain error types will be needed for any serious testing facility. This raises questions of research interest—which combination of tools provides the best coverage across types of errors? What are the criteria for integrating the tools (e.g., some tools might provide information which may be used as input to others)? How do we evaluate an integrated facility as opposed to evaluating individual tools?

An important consideration in the usefulness of a tool is the degree to which it may be incorporated in an average programmer's environment. Tools for data flow analysis, for instance, are easily incorporated in such an environment, as are some execution monitoring instrumentation facilities.

Tools are also language-dependent, i.e., most tools are designed for handling programs written in one language. Modifications will be needed to adapt them to other languages. But more important is the relation between tools and language constructs. For instance, while current data flow analysis techniques can handle most single-process programs, there is a need for new analytic techniques for dealing with concurrent-process programs. The synchronization constructs that characterize the latter introduce complex data and control flow possibilities.

It is useful to categorize currently available software tools into three classes—static, dynamic, and interpretive. Static analysis tools work on the structure of the program and do not involve execution. Facilities for data flow analysis and for gathering information such as cross-reference maps are examples of this type of tool. Program verification is also a static analysis technique. Examples of dynamic analysis tools include path generation routines and instrumentation for execution monitoring. The results of dynamic analysis are based on the performance of the program as it is executed for some inputs. On the other hand, symbolic execution—an interpretive technique—"executes" the program, but not for real inputs. Instead input is in symbolic form and the execution computes symbolic values for program and output variables.

Workshop presentations. One session was specifically devoted to test tools, while other workshop papers contained some discussion of experiences with various test tools.

A. Amschler from Karlsruhe, West Germany (co-authors are L. Gmeiner and U. Voges), presented her group's work in the design and implementation of an integrated testing system called SADAT (presumably an acronym for Static And Dynamic Analysis and Testing) for testing Fortran programs that have been compiled error-free. The main modules of SADAT are static analyzer, dynamic analyzer, test case generator, and path predicate calculator. The static analyzer produces several tables based on a simplified lexical analysis of the program source code and also generates a reduced program graph. Several types of errors can be detected at this stage, such as dead code, undeclared or unused labels and variables, and jumps into a loop. In addition, the output of the static analysis phase serves as a data base for later analysis.

SADAT's dynamic analysis documents the execution of program test runs. Basically this consists of instrumentation for the execution count of various branch points. A table is printed giving the relative and absolute numbers of executions and identifying those paths not executed during the test runs. This dynamic analysis is useful for identification of dead code, determining correctness of loop iterations, and optimization.

The test generation subsystem automatically generates a subset of paths with almost complete C-coverage (i.e., each arc and each node is represented in at least one path). In addition to the automatically generated paths, the user can specify a path as a sequence of statements.

The final module, not yet fully implemented, calculates path predicates by symbolic evaluation. The system is written in PL/I and runs on an IBM 370/168. There is a command language available for selective execution of parts of SADAT.

SADAT appears to be a well-engineered, habitable testing facility, with different tools integrated in a complementary manner.

L. Clarke presented the work of her group at the University of Massachusetts (co-workers are Neal Ogden and Daryl Winters) in the design of a system called Attest, to be used for symbolic execution of Fortran programs in the context of top-down testing. The Attest interface description language AID enables the user to describe both predicated and presumed relationships among program variables. This feature is important in top-down testing (or more generally, in testing using stubs for modules not yet written), since the specifications of the modules can be stated by means of AID commands, and symbolic execution can proceed as if the module is written and connected. AID has conditional execution constructs for the easy description of conditional procedure computations in early versions of a program. Fortran and AID can be freely mixed so that a module can be executed normally or symbolically. Attest also supports symbolic I/O. Using these features, the developer can produce successive refinements with progressively less AID and more Fortran code. For the class of applications where symbolic execution is useful, the Attest system can bring program creation and testing closer together and help realize the promise of stepwise refinement.

R. N. Taylor of Boeing Computer Services and L. J. Osterweil of the University of Colorado reported on their work in developing static and dynamic testing techniques for concurrent-process programs. This work was performed in connection with NASA-Langley's Must program, which addresses the production and testing of concurrent-process flight software.

Dynamic testing of single-process programs often includes generation of histograms. These describe a program's execution history by displaying counts of statement and branch point executions. Taylor and Osterweil proposed the notion of a process-state histogram as an extension of this technique for concurrent-process programs. Each time an event change takes place, a process state snapshot is made indicating the state of different processes. A series of such
snapshots is used to compute the process-state histogram. Automatic monitoring of system deadlock errors, which can occur in concurrent-process programs, can be incorporated by using several available algorithms. Dynamic assertion verification can be extended to concurrent-process software and is especially valuable to assure that scheduling and timing constraints are as designed.

Static analysis is often effective in weeding out errors that are costlier to detect by dynamic testing techniques. Extension of data flow analysis to concurrent-process software requires more complex control flow models. The PAF—process augmented flowgraph—is a concept designed to capture the data and control flows in concurrent-process programs with schedule and wait statements as synchronization constructs. The PAF and associated algorithms are capable of detecting errors due to shared data items being referenced by one process before any other process defines them. In addition, certain anomalies in the PAF indicate the occurrence of poorly coordinated processes. While PAFs are useful for a class of concurrent constructs, further work remains to be done for a broader class of synchronization constructs, such as open, close, and signal statements.

M. Holthouse and M. Hatch of Analytic Sciences Corporation discussed their experience with a set of tools including ones for static analysis, assertion processing, and test data generation for a path coverage based approach. While their experience indicated substantial benefits from the interactive use of these tools, they also discovered some potential problems. Sometimes protection schemes are devised to make each module “robust” against its environment. During integration, the protection schemes of two modules may overlap, causing some protection branches in the low-level module to become unreachable. Another issue is large system testing. Each module can be tested separately for high coverage, but when they are integrated discontinuities in overall system flow may be difficult to detect. For loop testing Holthouse and Hatch suggested that each loop be tested not only for its looping state, but also for a program flow not passing through the loop at all. While the testing tools they discussed cannot detect missing paths, Holthouse and Hatch pointed out that the close software inspection the approach forced them to make did in fact lead to the discovery of several errors of this type.

One of the accomplishments of the workshop was the establishment of a mechanism for the exchange of information on implemented test tools. Dr. Selden Stewart of the National Bureau of Standards has consented to be the coordinator of this activity. If you are interested, contact him at the National Bureau of Standards, Tech. A-265, Washington, DC 20234, (301) 921-3485. He will be preparing a questionnaire to obtain information about available tools, their language and machine constraints, documentation, and conditions of release.

With respect to future work in tool development, there is a real need for systematic evaluation of tools both in the context of testing programs in practical environments and in the context of carefully controlled experimental situations. This will yield insights about the relationship between error types, tools, and types of tasks. Some other aspects requiring attention include development of tools for a larger class of concurrent-process programs, incorporation of tools into the average programmer’s environment, and human factors in tool design. The last aspect is important because it is unlikely that software testing will soon become entirely automated. The human tester will continue to play a decisive interactive role.

Acknowledgment

I thank Dr. Lori Clark and Dr. Lee White for discussions helpful to me in preparing this report.

Test data generation: three approaches prevail

D. W. Fife
National Bureau of Standards

Test data generation could be defined simply as a collection of techniques for creating valid input data, considered in terms of feasibility, economy, and efficiency. But it is more important to discuss test data generation in the overall context of testing practice. Testing practice includes the testing methodology and tools that give the basis for test data generation. It also involves many other issues that affect evaluation and acceptance of data generation methods, such as testing administration, documentation, and auditing. For example, it is essential to be able to distinguish test cases and to accurately describe them relative to program specifications and testing goals. Test cases must be repeatable, particularly for real-time systems. Testing must be auditable and documented so that the software user or customer is assured that the proper tests have been applied and in the proper manner.

For the purpose of discussing test data generation, we can define testing as the execution of a program on finite input in order to infer conformance to specifications. Specifications agreed upon by user and developer are usually incomplete or else lack rigor and precision in many aspects. The user or the tester may need to augment or refine the specification so that the result of any test is precisely determined. The tester’s problem then is to use (possibly augmented) specifications along with accepted testing principles and his own knowledge to derive appropriate test cases.

Many workshop presentations touched on test data generation problems. For example, the research on program mutation presented by Fred Swayard of Yale and others aims to evaluate a programmer’s selected test cases, but does not define or prescribe how the test data is produced. Many tools such as SADAT (developed by Voges, Amschler, and Gmeiner of Karlsruhe, West Germany) include computer analysis of program predicates to help the programmer select test data to exercise all logical paths. But solutions of the predicates that would specify the needed data are not readily obtainable (and are undecidable in the general case), so the programmer must make his own analysis and frequently must choose test data by trial and error.

The software testing field still lacks one technique, or a set of them, that would be widely accepted as sufficiently effective to be a conclusive testing method. It is commonly believed that a minimum requirement is to test every program statement and branch at least once. This criterion is often marginal, but can be supplemented by other criteria as workshop participant Mark Holthouse of Analytic Sciences Corporation described. As a proposed minimum testing standard, this warrants a more extensive empirical evaluation. Sometimes, it leads to many more test cases than may be economically acceptable. Also, it is
not relevant to acceptance testing, which must be done from the program specifications rather than from the implementation or path structure.

Three approaches seem to prevail among the heuristic testing methods used today. In one, test cases are produced essentially to represent the normal or expected use of the program under test. This demonstrates to the user or customer that the program will perform its intended function. Test data are chosen by deciding what constitutes the most frequent or representative usage. If the application and specifications are suitable, a thread testing approach may be utilized, i.e., all normal functions, where small in number, may be exercised to show correspondence of output result to input data. 

Frank Woodall of Booz-Allen Applied Research discussed this technique as used for a weapons inventory management system done for the Air Force Armament Laboratory.

In the second approach, testing is done on a selective basis to expose errors under extreme or critical conditions. Kenneth Foster of GTE/Sylvania described such a method to derive error-sensitive test cases for conditional statements. Foster develops rather simple rules from an empirical analysis akin to traditional hardware logic fault analysis. The results suggest, for instance, a test of all ordering relationships (<, =, and >) between two variables regardless of the precise relationship demanded by the specification or program implementation. To reduce the number of test cases implied by these rules, Foster suggests testing only the "critical path," defined especially as a set of branches from program entry to exit which traverses the highest number of simple logical conditions.

The third prevailing approach to testing, which may be followed when the previous two approaches seem impractical or limited, is to invoke some notion of "coverage" of the program structure. Testing each statement and branch once is an example of this. Otherwise, this approach can be tied to some quantitative criterion which helps to guide the use of the available testing effort. Siba Mohanty of Martin Marietta Laboratories in Baltimore discussed some proposals of this kind. For example, one might test each program branch some number of times, the number being a function of the perceived complexity of operations performed on that branch. Random testing methods generally belong to this "coverage" approach.

The mortifying fact about software testing is that we have no conclusive measure of effectiveness for any testing method. At best, we have limited empirical data produced only recently, indicating the relative effectiveness of several methods on various types of errors. These studies are also among the few that have tried to formalize various testing methods as distinct procedures. From these results, and the limited guidance provided by present testing theory, it is apparent that no single method or concept will provide the answer to testing needs. Software developers and customers will have to develop strategies for applying different methods and criteria, taking account of the available testing resources and the nature of the software under test. In order to put such decisions on a sound footing so that reasonable properties of consistency, repeatability, auditability, and manageability will hold, future testing research must stress empirical studies of the effectiveness of well-formulated methodologies. Surveys and handbook-type articles which formalize and evaluate heuristic methods would be appreciated.

It is also clear that more research is needed toward a theoretical basis for software testing. A key goal should be the development of high-level specification languages serving both to convey design requirements and to automate the generation of test data. Arthur Duncan of Indiana/Purdue University at Indianapolis gave an example of how a test grammar derived from specifications might generate syntactically valid test data. A specification language must also be particularly strong in regard to both the application environment and the semantics of program execution. For testing support, it must also accommodate simple heuristic rules that will undoubtedly still be needed to overcome either the voids in theory or the high costs of testing.

References


Testing large systems: early planning essential

Alex Long
Electric Power Research Institute

The presentations in this session highlighted a number of the problems associated with the development, testing, and maintenance of large software systems. However, the lack of details about the merit and cost effectiveness of different software development and testing methodologies emerged as one of the major problems confronting the industry. This issue may account for the apparent technological gap between commercial practice and the research activities discussed in the other sessions.

First, are there a class of problems which are peculiar to or characteristic of large systems? In general, the development of such systems involves many people over long periods of time. Successful integration of these efforts requires careful coordination, scheduling, and configuration control. As was frequently pointed out throughout the meeting, testing cannot be left unplanned until the end. The functional requirements and design specifications must serve as the reference against which all testing should be conducted. Therefore, these documents have to be kept up-to-date. Last minute patches, which may seem expedient in the short term, will ultimately be fatal. For large systems, the high costs of maintenance over the product life (70 percent for some military systems) must also be recognized and planned for.

The first paper presented, by S. Oxman, discussed the Trident Command and Control Software System from the standpoint of the naval support organization responsible for final certification prior to use and for subsequent software maintenance. The complexity of the application (2M words of software), the large number of involved organizations (32 contractors), and the schedule constraints have all contributed to a very costly testing program ($400 per word for the system level software). The paper by C. Falcon describes the scheduling, documentation, and testing requirements adopted in order to optimize manpower and minimize schedules in a software project utilizing progressively more detailed "builds." The benefits were achieved by overlapping various phases of sequential "builds."
Both of these presentations acknowledge the problems previously singled out for large software systems. There don’t appear to be any sure remedies, but a number of promising and intuitively appealing concepts were discussed during the workshop. For example, the use of formal specification languages was mentioned as a means to (1) modularize software in a functional fashion; (2) increase overall testability; (3) facilitate early verification; and (4) maintain configuration control. The large percentage of errors attributable to misinterpretation of specifications, and the increased cost of corrections if errors are not detected early, supports the need for early testing during the preliminary software design phase. It was also suggested that the choice of programming language could have an important impact on software correctness and testability.

Once coding is complete, the large software systems developer must decide on the magnitude and type of testing effort. This is a difficult decision because there is so little information about both the effectiveness and cost of different testing methodologies and tools. A generally accepted tenet, however, is that a modular approach with a hierarchical testing plan (top-down or bottom-up) is desirable. The basic premise is that functionally independent modules or groups of modules can be developed. If this is correct, then the testing process can be decomposed into discrete tasks which may proceed independently. In addition, that the testing strategy and modification can then be limited to the test cases appropriate to the modified modules.

There appear to be three different criteria guiding the selection of test cases: (1) that based on functional requirements, i.e., representative of the intended application; (2) that based on program structure, i.e., coverage of all program segments; and (3) that based on possible error types which may be detected more readily by certain classes of data. The paper by M. Holthouse described the use of an automated test analyzer which measures branch coverage. This is representative of a class of dynamic testing aids whose prime benefit is felt to be that they focus the tester’s attention on all portions of the code. The system described requires only 20 percent overhead in both execution time and running space.

The testing of large software systems can be exceedingly difficult if many functions are involved and a large number of test cases are required. Simulated operation under normal, stressed, and degraded conditions for realistic periods is one of the preferred acceptance tests. Automated test case generation is another promising technique, providing that methodologies can be established for selecting significant test cases. However, it will also be necessary to develop automated techniques to facilitate the evaluation of the voluminous output for correctness against the original requirements. Redundant or diverse software could provide a simple means for making such an evaluation.

As a result of this workshop, however, it was clear that the software industry does not yet have a well-defined methodology or tools for testing large systems with a known cost effectiveness. In fact, a number of participants reported that even “structured programming,” as they had used it, was not satisfactory. This suggests that the testing community should sponsor R&D projects to provide better guidance concerning the applicability, limitations, and cost effectiveness of different methodologies. Realistic benchmark cases might be one way of documenting these quantities. In addition, the industry should adopt general metrics relating to software cost and correctness; these metrics should be documented in all large software application projects. This information would provide the testing community with invaluable feedback about the merits of their methodologies.

Finally, the participants felt that, once the benefits of particular testing strategies are established, the time delay until they are adopted can be reduced. To this end, they recommended that the testing community anticipate software industry trends (e.g., new languages) when developing new testing tools. They also noted that further involvement between theoretical activities and practical applications should be encouraged.

New approaches: mutation analysis elicits interest

Richard deMillo
Georgia Institute of Technology

As might be hoped for in a successful workshop on program testing, the new approaches were not confined to the formal session on “New Approaches to Program Testing.” However, participants seemed to be casting about for approaches which cut across the grain of current thought. Perhaps the key remark concerning new approaches to testing was that we should be on the lookout for techniques that exploit something significant about the programming process, and that have explicit (and potentially falsifiable) assumptions built into them.

This point first arose in a discussion of theoretical directions for testing. Consider, for example, the formal definition of the reliability of a data set \( D \) for a program \( p \) computing the function \( p^* \) with functional specification \( f \):

\[
\text{If } p^*(x) = f(x) \text{ for all } x \in D \text{ then } p^* = f.
\]

But in the above \( p^* \) is (formally) a completely arbitrary function. So any theorems derived concerning reliability are literally meaningless—they can be expected to hold under any interpretation of \( p^* \) in which some rather innocuous conditions are satisfied. Surely a statement which holds always cannot in itself deliver us useful information about the real world. Now of course it may be argued that what definitions like the above provide is a mathematical framework for analyzing the testing process.

That brings us back to “new approaches.” Are there any approaches to program testing that are not tautological? The difference between differential equations and physics is that the assertions about differential equations are valid while the assertions about the physical world (which may indeed be established using valid propositions of differential equations) are factual. They depend on a testable hypothesis—one whose truth is not guaranteed and which may be either confirmed or falsified by experiment.

A task we set forth at the start of the workshop was to identify new methodologies making use of specialized assumptions about programs and the programming task.

Lee White and Ed Cohen’s “Domain Strategies for Program Testing” illustrates the precision with which such a procedure can be carried out. White and Cohen set out six explicit assumptions for their domain strategy (e.g., the test cases do not exhibit “coincidental” correctness). The outcome of the White-Cohen collaboration is an elegant set of...
heuristics for determining path errors in programs. This is done by employing certain geometric intuitions about the manner in which the predicates of the various program paths partition the space of program variable values.

The three remaining contributions to the session all relate to the new program mutation methodology. Briefly, program mutation determines the adequacy of test data by indicating how many errors the data set is capable of distinguishing. Program mutation has a number of assumptions entering the theory:

- The competent programmer hypothesis: Programmers do not write programs at random. They write programs that are either correct or differ from a correct program in containing a small number of simple errors.
- The coupling effect: Test data sensitive enough to distinguish simple errors are also sensitive enough to distinguish more complex errors.

Both assertions are experimentally testable and much of the discussion of program mutations centered around experimental and other kinds of evidence for these hypotheses. The paper by Fred Sayward and Richard Lipton described the status of the program mutation research at Yale University, the Georgia Institute of Technology, and the University of California, Berkeley.

James Burns described an example of the experimentation possible with existing (i.e. prototype) mutation systems. Burns' hypothesis was that test data are largely transferable between different versions of the same program. Clearly, such an outcome would be desirable in the initial testing and during the update and maintenance phases of the software life cycle. By analyzing test data transferability between sorting programs and by using mutation as the basis for comparison, Burns developed an extensive set of test data. Unfortunately, the data resulting from the experiment seemed a trifle ambiguous with regard to the central hypothesis. However, there was clearly no methodological barrier to this kind of experimentation.

The final paper of the session, by Richard Lipton and Timothy Budd, dealt with theoretical insights drawn from program mutation and showed that for realistic models of computation, testing key correctness is feasible. These results seem to illustrate the directions in theory alluded to above.

Much discussion centered around the mutation approach. Is it a feasible technique? What is its complexity? What kinds of errors will it uncover? The liveliness of the discussion prompted the addition of a short session on program mutation. There seemed to be an underlying consensus that program mutation research is still too new to offer conclusive answers to these questions. Initial experience, however, has been positive—Budd offered the information that of the several hundred programs that have been subjected to and that have passed mutation analysis, none have been subsequently found to be in error.*

Representatives of the session's mutation group offered other investigators access to "small" versions of their prototype systems.

An interesting development with regard to mutation occurred when K. Foster reported on his "Error Sensitive Test Case Analysis" method of generating test data. Foster's method—which he apparently developed independently of the mutation research effort—is geared towards generating test cases yielding higher performance figures under mutation analysis. In experiments conducted by F. G. Sayward and A. T. Acree, Foster's test data was subjected to mutation analysis. Under two independent recordings of his test program into Fortran, Foster's test data had mutation levels of approximately 95 percent, meaning that all but approximately 5 percent of the simple errors which could have occurred in Foster's program had been accounted for by his test data. The remaining 5 percent were partly due to the fact that Foster's Pascal program had to be recoded into Fortran and partly due to the fact that Foster's test data inadequately tested his program for certain kinds of errors (which

---

*Subsequent to the workshop, Fred Sayward (Yale University) and Allen Acree (Georgia Institute of Technology) analyzed most of the sample programs and test data supplied at the workshop. They have reported that the mutation analysis results were consistent with the authors' expectations.
errors were made explicit in mutation analysis.

A significant point of interaction between the mutation group and other groups represented at the workshop was the arrangement of a large-scale experiment involving the mutation analysis of a 5000-line production Fortran code originally developed under government contract. It is hoped that this experiment will proceed in the next year and that the results can be reported formally.

In summary, the "new approaches"—with the notable exception of mutation—seem to be not so much new approaches as new perceptions of what should be attempted in program testing research.

Edward F. Miller, Jr., an independent consultant and lecturer, is technical director of Software Research Associates, San Francisco, California, a firm devoted to advanced computer technology and software testing technology, automated tool design and development, hierarchical design and implementation methods, and computer architecture. Miller received a BSEE at Iowa State University in 1962, an MS in applied mathematics at the University of Colorado in 1964, and the PhD at the University of Maryland in 1968, where he was also an instructor from 1964 to 1968. A member of the IEEE Computer Society, the ACM, SIAM and several honorary societies, he is currently editor of ACM Sigarch's Computer Architecture News, and an associate technical editor of Computer.

S. L. Gerhart is a member of the research staff of the USC Information Sciences Institute, Marina del Rey, California. Her research interests are in the area of program verification, including the theory and practice of proving correctness of programs, and the development of programming methodologies that reduce the difficulties of verification. Gerhart received the BA degree in mathematics from Ohio Wesleyan University in 1965, the MS degree in communication sciences from the University of Michigan in 1967, and the PhD degree in computer science from Carnegie-Mellon University in 1972. She is a member of ACM, the IEEE Computer Society, and Sigma Xi.

William E. Howden is an associate professor of computer science at the University of Victoria in Victoria, Canada, where he is on leave from the University of California, San Diego. Howden has carried out extensive research projects in both the practice and theory of program testing. He has conducted professional seminars on software validation in the United States, Canada, and Great Britain and has published over twenty papers on a wide variety of different topics in program testing. A past consultant to branches of the US and Canadian governments and to large industrial organizations, Howden holds the PhD in computer science from the University of California, Irvine.

Herbert Hecht is president of SoHar Incorporated, an organization engaged in studies and consulting in computer software and hardware reliability. He has published many papers on computer and control system reliability and has conducted a short course in software reliability at UCLA. Hecht holds a BEE from the City College of New York, an MEE from Brooklyn Polytechnic Institute, and a PhD in engineering from UCLA. Registered as a professional engineer (control systems) in California, he is vice chairman of the Computer Society TC on Software Engineering and has initiated efforts to standardize software terminology and practices.

B. Chandrasekaran is a professor of computer and information science at Ohio State University. His current research interests are artificial intelligence, pattern recognition, computer graphics, and computer program testing. Chandrasekaran received his Bachelor of Engineering degree with honors from Madras University in 1963 and his PhD from the University of Pennsylvania in 1967. A vice president of the IEEE Systems, Man, and Cybernetics Society in 1975, he is currently chairman of that society's Technical Committee on Artificial Intelligence as well as associate editor of IEEE Transactions on Systems, Man, and Cybernetics.

Dennis W. Fife is chief of the Computer Science Section at the National Bureau of Standards, where he directs projects in software management and quality assurance, data base software evaluation, and software testing. Prior to joining NBS, he worked with the Mitre Corporation on design and evaluation projects for various military systems. Fife received a BSEE from Michigan State University and a PhD in electrical engineering from the University of Michigan. He has been active in software engineering activities as a member of the Computer Society's TC on Software Engineering.

A. B. Long is currently a project manager in the Nuclear Safety & Analysis Department of the Electric Power Research Institute, Palo Alto, California. His responsibilities include the formulation and management of research and development projects involving the application of advanced technologies (hardware and software) to the control and safety of nuclear power plants. Long holds a BA in physics from Williams College and the PhD in nuclear engineering from the University of Illinois.

Richard deMillo is an associate professor of computer science at the Georgia Institute of Technology. A technical consultant to several government and research agencies and to private industry, he is interested in the theory of computing, programming languages, and programming methodology. DeMillo received the BA in mathematics from the College of St. Thomas, St. Paul, Minnesota, and the PhD in information and computer science from the Georgia Institute of Technology. He is a member of ACM, the American Mathematical Society, AAAS, and the Association for Symbolic Logic.