NRL Invitational Workshop on Testing and Proving: Two Approaches to Assurance

Georgetown University
Washington, D.C.
July 9-11, 1986

Organizing Committee

Carl E. Landwehr, NRL John McLean NRL
Susan L. Gerhart, MCC Donald I. Good, U. Texas

Nancy Leveson, U.C. Irvine

Overview

The Naval Research Laboratory sponsored this workshop to invigorate research in both program verification and program testing through cross-fertilization, to document the state of the art and practice in both areas, and to identify current assurance requirements and techniques for meeting them. Approximately 50 invited researchers and practitioners participated over a 3-day period. The workshop was held in conjunction with the COMPASS 86 conference, and the initial and final days of the workshop were open to COMPASS attendees.

Tutorials characterizing the current state of testing and proving techniques and identifying industry and government assurance requirements occupied the first day of the workshop. These provided a common base for five discussion groups held during the second day. The discussion groups addressed (1) the role of specifications in testing and proving, (2) hybrid approaches of testing and proving, (3) levels of assurance, (4) interactions between testing/proving and software engineering, and (5) cost effectiveness. The leader of each of these groups summarized the discussions and conclusions on the final morning. Harlan Mills of IBM then provided a critique of these results. Amrit Goel of Syracuse provided an impromptu closing talk on alternative statistical models for software testing

The accompanying summaries were written by the discussion group leaders following the workshop as a means of documenting our results and circulating them to a wider audience. Although the leaders have tried to record faithfully the results of the discussions, these summaries (and this preface) have not been reviewed or approved by the other participants of the groups. In addition, Dr. Mills has provided a note on his thoughts about the workshop topics.

Comments

An interesting fact that cannot be gleaned from examining the group summaries is the popularity of each group as determined by the number of participants requesting to be in that group. For example, we had originally planned for a sixth group on domains of applicability, but could find no takers, and the group on cost effectiveness was not popular among those participants who

expressed a group preference. Perhaps the two phenomena are related since a major factor in determining whether an approach is useful in a certain domain is its cost effectiveness in that domain. In any event, the concept of a domain of applicability for each approach arose often in the groups.

To get a tighter grip on this, consider the distinction made in various ways in several of the discussion groups between *verification*, where one establishes the consistency of one formal object (e. g., a program) with another (e. g., a specification), and *validation*, where one establishes the consistency of a formal object (again a program) with an informal one (e. g., a user requirement). Proving is all but impossible in the latter domain, while testing, though helpful in both verification and validation, seems particularly well-suited for the latter. Yet the conclusion that we should prove what we can and test what we can't prove is premature. Testing can sometimes provide cheap verification. For example, athough for peace of mind we may want to prove that a certain program state is unreachable, testing can satisfactorily demonstrate that a certain program state is reachable.

If the view that testing and proving are complementary methods of attaining assurance emerges from the group summaries, the view that they share much is also present. Formal specifications, though not a prerequisite for testing, are useful in generating oracle programs. Further, well-modularized programs are aids to both proving and testing. Similarly, though the discussion groups found much that we know about proving and testing, they also recognized much that there is much that still needs to be learned about each. One such issue that was partly addressed in Goel's discussion of statistical testing is determining the point where running more tests ceases to be cost effective. Presumably, the same question can also be asked for proving.

Acknowledgments

NRL would like to thank, first, the participants for their willingness to share their knowledge, to broaden their own horizons, and to pursue candid and vigorous discussions on the topics of the workshop. We are particularly indebted to the tutorial presenters: Richard Platek, Bill Pase, Don Good, Steven Zeil, and Richard Taylor; to the discussion group leaders, and to the members of the requirements panel: K. Speierman of NSA, William Wilson of NASA, W. E. Ford of Oak Ridge National Laboratory, Dres Zellweger of the FAA, and Nancy Leveson. John Cherniavsky of Georgetown helped out with several last-minute crises, and we are grateful to LT Greg Johnson and the others on the COMPASS 86 conference committee for their assistance with the mundane but crucial details of local arrangements.

NRL Invitational Workshop on Testing and Proving, July, 1986

Discussion Group Summary:

Role of Formal Specifications

Chair: Jeanette Wing, Carnegie-Mellon U.

Participants

Dan Craigen, I.P. Sharp
Diane Britton, RCA
Steve Crocker, Aerospace
John Gourlay, Ohio State U.
Ann Marmor-Squires, TRW
DeWayne Perry, AT&T Bell Labs
Frieder von Henke, SRI
Steven Zeil, U. Mass

Our group had a balanced representation from academia and industry, a total of two from the testing community and a half dozen from the proving community. We focused our discussion on the role of formal specifications in software development; the current use of specifications in practice; and the future roles of specifications, all with respect to proving and testing.

1. Roles of Formal Specifications for Proving and Testing

In software development, formal specifications can be used for design, documentation, verification, proving, and testing. Ideally, writing formal specifications should guide the software development process. Indeed, it is the act of specifying that often is more beneficial than having the end product of specifying, i.e., the specification document itself. The roles of formal specifications for design, documentation, and especially for verification are well-known; their use is justified and documented in literature. Thus, we concentrate here on the role of formal specifications for proving, of which verification is a special case, and testing.

Proving, in a broad sense, is an activity during which one proves properties of a formal, i.e., mathematically meaningful, entity. For example, one proves a program correct or that sets have no duplicate elements. Verification, in a narrow sense, is the activity during which one proves correctness and consistency properties of one formal entity with respect to another formal entity. Verification necessitates having a formal specification, whether one is verifying a top-level specification, low-level code, or some entity in between. Thus, we distinguish verification from validation where in the latter activity one checks for whether a formal entity (*e.g.*, a specification or program) satisfies an informal entity (*e.g.*, a client's set of informally stated requirements).

Testing also necessitates having specifications, though not necessarily formal, as evidenced by the current practice of using informal or semi-formal specifications to generate test cases. Any kind of specification, such as operational

specifications, can be used to formulate testing criteria. Formal specifications can help narrow the domain of cases to be tested. They can reduce the input space to a testable subspace. If a formal specification is executable, or at least evaluatable, it can be used for writing automated oracles. Given the output of a test case generator and the output of executing a program over a test case, an automated oracle can tell whether an error is present in the program. Tools used for verification such as automated theorem provers and rewrite-rule engines can thus also be useful for testing. Testing generates the same kinds of theorems as proving; they tend to be simpler, but in greater abundance. Typically, testing requires determining whether two expressions are equal or not. Instead of trying to prove a handful of verification conditions, however, one is faced with hundreds of expressions to compare or evaluate. Finally, an executable or evaluatable specification itself is subject to current testing techniques. These uses of formal specifications for testing suggest that testing, as for proving, can and should occur early in the software lifecycle.

Testing and proving are complementary. Testing is necessary for validation because of the informal entity involved. Some properties, such as real-time behavior, are hard to capture in a formal specification and then verify, but easier to test for. Consider a user interface for which an icon is dragged across a bitmap display. A likely requirement of the system would be that the appearance of the moving icon be smooth or continuous. Such "user-friendly" behavior is hard to state formally. Whereas testing can help prove that certain states are reached (liveness), however, some classes of formal specifications are more helpful in proving that certain states can never be reached (safety). Of course, testing is insufficient for life-critical software.

2. Current Use in Practice

Formal specifications used in practice are almost exclusively for software design, verification, and documentation, and not for testing.

The majority of practicing software engineers in industry use, if anything, high-level design languages such as PSL/PSA, SADT, Jackson's method, or an in-house semi-formal design method. The security sector, largely supported by government contracts, provides the largest community of formal specification users. Industrial research laboratories such as at Aerospace Corporation, IBM, AT&T Bell Labs, MCC, and DEC, have explored, applied, and gained valuable experience in formal specification languages, tools, and methods. These experiences seem to represent a few isolated success stories. Academia has been doing steady research in the area and spreading its ideas into the industrial community. For example, the Gypsy verification system (University of Texas, Austin) now exists at 25 sites in the U.S. and Canada.

Outside of North America, in particular the United Kingdom, Denmark, Germany, and France, awareness and use of formal specifications is more common. A combination of cultural , economic, and historical reasons may explain this difference. The Vienna Definition Method (VDM) and Z are two examples of

formal specification techniques being taught and used (though not heavily) in industry. VDM, for example, has been used to help define the semantics of Ada which in turn was used to help develop an Ada compiler.

3. Future Roles

Testing can benefit from more use of formal specifications and from exploiting technology that has resulted from a decade of verification research such as automatic theorem provers. Testing can also benefit from making more use of data type semantics and modular structure of large programs. Research in formal specifications has to go beyond the specification of functional correctness and security aspects of systems. In particular, specifying and proving performance constraints for real-time systems, and specifying and proving the recoverability of data of transaction-based distributed systems are open problems. These problems require fundamental research, as well as good engineering to make the theoretical results palatable to practitioners.

In order to increase the use of formal specifications, we need more production-quality tools that are packaged properly; more educating of managers and software engineers; more convincing, publicly accessible, written examples. Finally, it is the responsibility of those who advocate particular specification languages, tools, and methods to inform the user as to what specific uses the language, tool, and/or method are suited for. A user must be aware of a particular technique's limitations as well as its benefits.

NRL Invitational Workshop on Testing and Proving, July, 1986

Discussion Group Summary:

Hybrid Approaches

Chair: Susan L. Gerhart, MCC

Participants

John Cherniavsky, Georgetown U.
Paul Garnett, NSWC-Dahlgren
Mike Gorlick, Aerospace
Larry Hatch, DoD
Nancy Leveson, UC Irvine
Bill Pase, I.P. Sharp
Rami Razouk, UC Irvine
Deb Richardson, U. Mass.
Bob Westbrook, NWC-China Lake

1. Group Mission and Composition

This group was charged with finding potentially useful combinations of testing and proving approaches. (Another class of hybrid concerns -- combining methods within testing and within proving -- is worthy of study, but was not the focus of the working group.) Perspectives represented by individuals within the group were: experience with large system testing, building and using proving systems, building and using Petri net models, defining and using safety models, and defining and evaluating testing methods.

After justifying further consideration of hybrid approaches, several combinations of approaches were suggested and analysed, resulting in a collection of problems to be solved.

2. Reasons for and against hybrid approaches

Why consider hybrid approaches? Here were the group's initial reactions:

1. A subjective basis lies in experience: it does make you feel more assured to know that both testing and proving methods have been applied to critical software systems, such as flight control. Would you step on a flight vehicle whose software had only been proved? Few of us would, but the first shuttle launch-computer synch problem reminds us of the deficiencies of testing. Why do we feel this way? A related subjective conjecture is that anything that makes you think more about your program is bound to improve it.
2. More objectively, no one technology can cut it today. Technology for proving properties of systems is maturing and becoming industrialized in security applications, while considerable testing technology has already found its way into industrial use. But reliable, production use of either (comparable to our current use of and trust in compilers) is not imminent.

3. Each approach is based on different assumptions about how the world works, e.g. proving assumes the world can be axiomatized and testing assumes the world is continuous.
4. Practitioners of the approaches experience wholly different viewpoints. It is widely believed that multiple viewpoints is a key to finding flaws in anything. The approaches may be used to complement or to check each other.
5. While testing must be based on specifications, the use of proving encourages more formality in those specifications.
6. The approaches appear to apply differently in different phases of the life cycle, e.g. operational testing appearing necessary for acceptance and deployment with proving being both easier and more effective in early phases. Also, a top level specification may be proved internally consistent but can be verified only by testing to determine if it behaves according to our informal, internalized expectations.

Are there any reasons against using hybrid approaches? Yes:

1. A hybrid approach may divide the resources available. Furthermore, the resulting management tasks could be greatly complicated, e.g. maintaining traceability of both testing and proving data with respect to a system.
2. Testing has worked well enough in the past, so why not perfect it? Similarly, proving is beginning to work well, e.g. in Harlan Mills' IBM Cleanroom approach, so why not perfect that? Or if the separate approaches aren't working well, maybe something other than the opposite approach is needed, e.g. inspections and reviews (which the group decided fell into one of the two approaches anyway, differing only with more emphasis on people than on technology).
3. We don't know how much overlap there may be in using both approaches. Redundancy could be beneficial, but waste must be avoided.
4. There may be an eminently testable or provable language that could ameliorate the problems of one approach.

While there are good reasons against hybrid approaches, these reasons can be turned into problems to be solved (see section 4).

3. Combinatorial Alternatives for Hybrid Approaches

It is not hard to conjure up a variety of combinations that might be useful. The following table provides the structure for covering many combinations:

	P1	P2	...	Pn
Design	t	t&p		t v p
Code	p	t		p

That is, the major life cycle divisions to be considered are *designing* (the production of a description of what is to be implemented to satisfy requirements)

and *coding* (the production of the implementation), since their intellectual activities and technologies differ. There may be many ways of filling in such a table of approaches, using combinations between and within levels. Here "t" or "p" denotes some known testing or proving method, respectively, e.g. a testing coverage method or an inductive proving method.

The more significant part of this table is the division into P1, P2, Pn, which denote any of the following:

Properties:	Qualities that we think of as safety, correctness, performance, etc.
Parts:	Structural parts of a system, e.g. modules, subsystems, or slices
Persons:	The people judging or providing the necessary qualities, e.g. buyers, government security certifiers, quality assurance groups
Processes:	Behavioral aspects of a system, e.g. dispensing of money from an ATM
Phases:	Temporal divisions of software production activities, e.g. acceptance planning, design generation

Ideally, an assurance plan for a project could be written using tables such as these for all the relevant players, system decompositions, important properties, etc. Being realistic, the above tables will always contain at least one *t*, for testing in the operational environment.

There are other hybrid approaches. An intuitively obvious, apparently universal combination is to use one approach quickly followed by full-blown application of the other, e.g. testing a program to make sure it executes on some data before attempting a proof or doing a 5 minute mental proof to assure that there is some reason for believing a program to be tested is correct. Another general approach is to use one method strongly to structure a program then using the other to provide complementary verification information. For example, the use of data and control invariants may structure a program while coverage testing may be applied in detail.

A more technical approach is the use of equivalence classes to link tests and proofs, e.g. an inductive proof that all data in a class will be treated similarly by a program while only one member of the class need be tested. Such a process could be driven by either approach, using equivalence class proofs to reduce the overall effort of proofs in special cases or using them to reduce the number of tests that must be made. Yet another hybrid approach is to perform proofs that leave assumptions to be tested in the operational context. Other possibilities include symbolic execution (using symbolic test data to generate path descriptions which are subjected to simplification using proving methods) and generating test predicates from specifications.

The key task, now, is to characterize the domains of applicability: which software production models fit the above profiles, which kinds of applications are best matched to which combinations of approaches, and which methods provide the greatest assurance to which classes of people needing assurance. It was rapidly found to be beyond the capability of the working group to perform this task in one day. Indeed, it proved to be difficult to find *any* examples where there was a clear characterization of applicability, e.g. where testing was clearly easier than proving. Even such examples as finite state machine based programs had arguments both ways about what was hard and easy. It will be necessary to better characterize the requirements and assumptions of the testing and proving approaches before attempting to characterize applicability in specific situations.

4. Problems to be Solved

The following problems were identified by the working group. The list is hardly complete, but represents typical problems to be solved by researchers to push further the possibility of hybrid approaches.

4.1. Theory

1. Precisely define the connection of equivalence classes with testing and proving methods. Where is this approach effective?
2. Characterize the information content of each approach. What is learned from testing (what issues are resolved, what facts are obtained)? ditto proving? Formulate the information overlap. What is useful and what is wasteful redundancy? What information is missing?
3. Correctness is a well-defined property in most cases. Find good definitions for other properties of interest: reliability, security, and so on.
4. Develop reusable domain theories (*e.g.*, the theory of paging in an operating system) to reduce proof over-head.

Of these, problem 2 is new and central to hybrid approaches, while the other problems are recognized and being worked on.

4.2. Methodology

1. Assuming executable specification languages become available: how will they be tested? how will they support proving? how will properties be distributed between testing and proving?
2. The hybrid approach of reducing proofs to environmental assumptions raises the important problem of how their testing will be performed in the operational environment.
3. Following the templates described for assigning testing and proving methods to designs and code, find the proper divisions of labor. In other words, characterize the domains of applicability of the approaches, taking into consideration their roles in a hybrid context.

4. Study the ways that testing and proving methods may be systematically applied in a hybrid context: which orders will be pursued? how will information flow?

4.3. Technology

1. Develop tools that integrate testing and proving approaches.
2. Find support for traceability and version control in a context of hybrid verification.
3. Find programming language semantic manipulators that work for both testing and proving.

4.4. Measurement

1. Devise experiments to ascertain the value in practice of hybrid approaches.
2. Develop a cost model for determining allocation of testing and proving methods within the templates.

5. Conclusions

The most significant development of the working group was the number of combinations of approaches that appear potentially fruitful, *i.e.*, could not a priori be excluded from further consideration. Characterizing domains of applicability for hybrid approaches appears quite challenging.

A balanced view of the potential of hybrid approaches must be maintained, since attractive theoretical possibilities may be offset by increased difficulty of management. Many well-studied technical problems could be solved or ameliorated in a hybrid context, but new technical problems are readily apparent.

NRL Invitational Workshop on Testing and Proving, July, 1986

Discussion Group Summary:

Levels of Assurance

Chair: Rich DeMillo, Georgia Tech

Participants

Debbie Cooper, SDC
Paul Fairfield, U. Liverpool
Amrit Goel, Syracuse U.
Bret Hartman, RTI
J. C. Huang, U. Houston
Bill McKeeman, Wang Inst.
Andy Moore, SA&E

1. Motivation for Panel

The panel was in general agreement that the primary motivation for considering the relative levels of assurance provided by testing and proving methodologies was the need for increased quality in software products and software intensive systems. This need is expressed in several ways, including:

(1) the numbers of reported defects in software systems, and

(2) the increasing functional requirements on software systems.

Testing and Proving are both designed to enhance levels of assurance about software product quality, but what *exactly* does that mean? In particular, the following questions seemed to be relevant:

- Who is being assured?
- About what?
- What constitutes a "level" of assurance?

A goal of the panel was to construct a set of *strawman* assertions such as

"Proving is good at.....in practice."

"Proving is bad at...in principle."

Testing may be good at...someday.",

where the "..." were to be filled in with definite statements about the kinds of assurance levels that could be achieved.

2. Problems

The panel did not achieve its goal. There were three *problems* that had to be addressed in the brief time the panel had to isolate some issues.

At first blush, the first problem seemed a surprising one. It dealt with terminology, of all things. There were, of course, the usual terminological wranglings over what "verification" means (it means demonstrating consistency with an immediately preceding specification) and what "validation" means (it means

demonstrating satisfaction of a user requirement). There was also a set of terminological issues that weren't so easily settled. In hindsight, these issues addressed the second question: what is being assured?

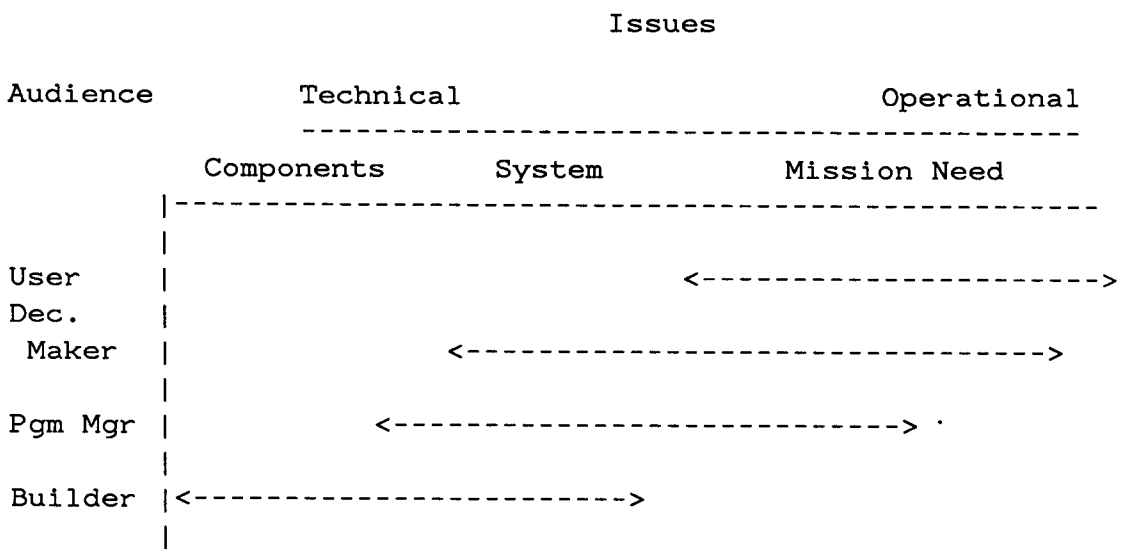
In the theory of inductive logic (cf. C. G. Hempel, *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science,*, Collier-MacMillan Ltd, London, 1965 -- particularly the essay "Confirmation, Induction and Rational Belief") the following problem arises. Suppose we are trying to assess the degree to which a piece of evidence E confirms a (scientific) hypothesis H. The extent to which E confirms H may refer, on one hand, to the extent to which E satisfies some objective criteria that confirmations of H are supposed to satisfy. On the other hand the degree to which E confirms H may refer to the extent to which E reliably assesses the truth of H and inferences drawn from H. Logicians have resolved these dichotomies in imperfect ways. Interestingly, these seem to be the same problems that arise in level of assurance for software: the first kind of confirmation is really the verification of some specification and the second is a kind of validation.

The second problem concerned the relatively incompatible points of view held by developers of commercial software and developers of Defense software. In the latter case, the process of software development is highly managed. Requirements tend to arise from a well-defined user community ("...in order to maintain a sortie rate of ___ against a specified threat in the presence of a logistics delay of ___, a fleet of ___ aircraft, each capable of ___ hours of mission flight without critical failure is needed..."). Overall development-acquisition-deployment decisions are in the hands of identified decision-makers who gather data and evaluations aimed at reducing the risk of faulty decision-making. A program manager gathers, monitors and directs the human and technical, and economic resources needed for the development. Finally, the technical designers-builders actually construct and test the system. In the case of commercial software development, there is considerably more variation. The development process may be as highly structured as indicated above. On the other hand, commercial software is frequently developed by one or two programmers, working from informal (sometimes non-existent) requirements specifications, and without clear lines of authority for decision-making. As development proceeds from highly structured environments to less structured ones, the need for formal assurance mechanisms becomes less pronounced. In addition, a separate panel of the present workshop (the "requirements panel") identified *large systems* as the target for the workshop. Accordingly the panel agreed to concentrate on the Defense software problem.

The final problem centered around the dichotomy between economic and technical matters. Specifically, the issue to be addressed was whether or not economic limitations and concerns were sufficiently technical and well-defined to warrant consideration by the panel. The answer seemed to be a resounding "yes".

3. Audiences versus Issues

No uniform criteria exist for relating the "Who" to the "About What" components of assurance. For a given software development effort, the issues to be resolved relate to different *audiences* as illustrated below.



The technical issues are those questions that must be addressed to adequately assess engineering quality. In hardware, these issues may involve such criteria as machining tolerances. In software the technical issues may question generic characteristics such as whether certain coding standards have been followed or specific characteristics such as whether or not a certain specification has been satisfied. Operational issues, on the other hand, are those questions that are raised to provide insight into operational suitability and effectiveness. Typical examples of operational issues are the extent to which the system performance envelope satisfies mission needs and whether overall system availability is sufficient to ensure completion of mission objectives.

4. Levels of Assurance

Assurance relates observational or other information (e.g., a correctness proof) about software to a set of issues. This relationship is generally established by specifying a property Q that -- when satisfied -- resolves the issue. Examples of such properties, with respect to a software system S, are the following:

- o S has branching complexity less than x
- o S is consistent with specification statement D,

- o S achieves an operational reliability of at least R(t)
- o S achieves performance parameter P
- o S meets user expectations.

A *level of assurance* represents a degree of confidence that one of these issues has been resolved. Equivalently, a level of assurance is a degree of confidence that one of the specified properties obtains. There are three alternative definitions that arise naturally:

Definition A: The degree of confidence that the software behaves as intended.

Definition B: For specified property Q, the degree of confidence that the software satisfies Q.

Definition C: For specified property Q and person P, the degree of confidence possessed by P that the software satisfies Q.

All three definitions appear to be important, although they are increasingly less tractable insofar as they apparently demand increasingly specific assurances.

The definitions, however, leave open an important problem -- one which is, on the surface, as difficult as the original: What is a degree of confidence?

To give an operationally meaningful definition, three different properties Q need to be distinguished. Actually, the (single) property Q, mentioned above, resolves into three Q's (which we will call Q(I), Q(A), and q(A)) that are needed to provide a sensible answer to a technical or an operational issue.

Q(I): This is the ideal or the required property.

Q(A): This is the actual property possessed by the system

q(A): This is the property that we are able to infer that S possesses by attempting to observe (or infer) Q(A).

The "degree of confidence" mentioned in Definitions A-C is simply the odds we are willing to bet that the difference between the ideal property Q(I) and the measured property q(A) is *really* the difference between the ideal property and the true property of the system.

The notion of "betting odds" is an important one. It suggests a connection between these definitions and economic concepts such as cost versus benefit, cost versus risk, and cost-of-error versus level-of-effort. It is also sufficiently loosely defined to admit many interpretations. In some circumstances (e.g., when there are stochastic elements that can be described or controlled) betting odds may, in fact, be statistically meaningful probabilities. In other cases, betting odds may reflect subjective judgements. In still others, they may represent mathematical assessments of validity or conditional validity.

The "ideal" property is an important aspect of the problem. It may seem natural to factor it out, however, doing so may turn the problem of assessing levels of assurance into an operationally meaningless problem. Suppose that "properties" are really quantitative parameters, so that the degree of confidence corresponds to the odds that

$$Q(I) - Q(A) = Q(I) - q(A).$$

It may seem natural to repair this equation as follows:

$$Q(A) = q(A).$$

This doesn't always work, since a convincing level of assurance in the first case may not be convincing in the second (these paradoxes involving logically equivalent properties are well-known in logic and may be found, for example, in Hempel's essay, cited above).

Finally, the issue of "Who" can be incorporated or addressed in these definitions as appropriate.

A summary conclusion -- maybe an obvious one that did not require Definitions A-C -- is that, whatever we mean by levels of assurance, these levels are not totally ordered. There will be incomparable levels as well as long and short chains. A theory is needed to establish some basic properties of the ordering.

5. Establishing the Ordering of Levels -- An Agenda

One approach to providing an ordering of degrees of confidence is to identify a set of assurance functions, the issues and audience they address, and the extent to which a given methodology exhaustively implements the function. Examples of such functions include the following:

- find errors
- demonstrate operational performance
- validate software against informal user expectations and needs
- validate software against a statement of user requirements
- verify the software against a given specification
- validate a specification against informal user expectations and needs
- validate software against a statement of user requirements
- verify a specification against a given (prior) specification
- exhibit a given static property of the software
- ensure adherence to a given development approach
- certify the software against some (pre-existing) standard
- demonstrate the modes and effects of software failure

For example, verifying the program addresses the correctness property and is primarily of importance to builders and the program managers. While a valid proof provides high confidence in correctness, the extent to which testing provides degrees of confidence is a subject of current research. On the other hand, failure modes and effects analyses are of critical importance to users. Failure modes are exhibited by testing at levels that have not really been determined with great precision. Failure modes and effects are, by definition, not exhibited by proving so proving can give no confidence in this area.

NRL Invitational Workshop on Testing and Proving, July, 1986

Discussion Group Summary:

Software Engineering Interactions

Chair: John Gannon, Univ. of Maryland

Participants

Susan Dart, SEI
Ben DiVito, TRW
Paul Eggert, SDC
Stuart Faulk, NRL
Richard Kuhn, NBS
Peter Neumann, SRI
Richard Smaby, MITRE
Dick Taylor, UC Irvine

Our discussions focussed on the impact of testing/proving considerations on the structure of software systems and on other software engineering practices and goals. We also tried to assess current practices to determine what kinds of properties were likely to be demonstrated using the respective technologies, and organizational issues relating to testing/proving.

It is obvious that good system design makes both testing and proving much easier. Program verification benefits both from layers of abstraction and from modular decomposition within the layers. In the verification of a hierarchical system, we demonstrate that lower-level layers correctly implement higher-level layers until a layer is reached that can be executed on a target machine. The series of verifications reduces the complexity of the mapping function between states of the implementation layer and those of the highest layer design. Within each layer, modularity makes specifications easier to write and keeps proofs smaller. More detailed discussions of these issues can be found in [1-4]. Testing places smaller demands on system structure than does verification, but still benefits from hierarchical and modular decomposition. Although intermediate layers in a hierarchical design may not be executable on the target machine, they may still be used in symbolic executions. Modularity makes it possible to execute a system component over more of its domain than might be exercised by an acceptance test, perhaps resulting in more robust behavior when the system is operational.

We discussed the possibility that designing for easy testing/proving might negatively impact other software engineering goals (e.g., ease of change, maintainability, etc.), or affect other software engineering practices. Of the goals, we concluded that only efficiency might suffer. While it is possible that excessive layering or modularity imposes run-time overhead, the implemented version of the system need not be identical to the verified version. Correctness-preserving transformations (e.g., inline expansion of procedures) can be applied to increase efficiency. Another issue raised was the possibility that a less efficient design

alternative would be selected in order to reduce the difficulty of its proof. We felt that the relationship between testing/proving considerations and software engineering practices was mutually beneficial. For example, we discussed how rapid prototyping might be used to refine and test specifications.

In discussing what kind of properties were likely to be amenable to testing/proving, we concluded that each type of validation technique had a role in the software engineering process. Properties that are defined in terms of avoiding enumerable anomalies (e.g., downward flows of information in multi-level security models, safety, liveness, etc.) were good candidates for verification. More complex properties (e.g., the function computed by a unit of code) could either be tested or proved, but that both techniques relied on the presence of a formal specification for the function. (Even program testers agreed on the usefulness of a formal specification in recognizing errors.) Properties that were not generally specified formally, but which could be consistently judged by human oracles (e.g., system performance, the behavior of machine arithmetic, etc.), were more likely to be tested than proved. One final category of properties emerged under the title user expectations" (things that are inconsistently judged by human oracles such as user-friendliness"). Neither validation technique seems to be much help as long as these properties remain so ill defined.

Finally, we considered organizational issues relating to the use of testing/proving techniques. Currently industry makes little use of either verification or the more formal aspects of testing (e.g., test oracles, symbolic execution, or even test coverage metrics). This situation exists because budgets are set for producing products rather than for developing tools or training personnel to use new methods or tools. A manager seeking to produce a system within time and money constraints is unlikely to dilute team effort producing new tools, documenting tools his team developed for their own project, or training his team to use a new method or tool. Only changes in the economic basis for decision making (perhaps in response to technical breakthroughs in component reusability) are likely to bring changes in this behavior. Software reuse can raise both product quality and programmer productivity, making verification viable. We discussed the central role that formal specifications are likely to play in software reuse, increasing programmers' confidence in what software components do and decreasing their eagerness to rewrite a piece of software they cannot understand. We also recognized that inadequate mathematical education is likely to make programmers reluctant to try either verification or testing based on formal specifications.

References

[1] L. Robinson and K.N. Levitt. Proof techniques for hierarchically structured programs, *Communications of the ACM* 20, 4, (April 1977), 271-283.

[2] J.M. Spitzen, K.N. Levitt, and L. Robinson. An example of hierarchical design and proof, *Communications of the ACM* 21, 12, (December 1978), 1064-1075.

[3] M. Melliar-Smith and J. Rushby. The enhanced HDM system for specification and verification, ACM SIGSOFT Software Engineering Notes 10, 4, (August 1985), 41-43.

[4] P.G. Neumann. On hierarchical design of computer systems for critical applications, *IEEE Trans. Software Engineering,* 12, 9, (Sept 1986).

NRL Invitational Workshop on Testing and Proving, July, 1986

Discussion Group Summary:

Cost Effectiveness

Chair: Donald I. Good, U. Texas

Participants

Mark Cornwell, NRL
David Hedley, U. Liverpool
Joseph Kmiecik, Grumman
David Musser, GE
Richard Platek, Odyssey Res. Assoc.
Vinceent Reed, Teledyne-Brown
Barry Stauffer, Logicon

I must begin this report by apologizing to the other members of this group for not being able to attend the session which I was supposed to chair! I was not informed of the situation that caused my absence until the afternoon before the session was scheduled to be held. I, therefore, would like to thank and commend all of the participants in this session for their efforts to address some very difficult issues and to come forward with some constructive ideas.

One of my current interests is discovering what kind of hard, objective justifications, if any, can be made for the cost-effectiveness of formal verification. To put it in commercial terms, how do I persuade a manager that verification might reduce some important costs? The search for an answer to that question has lead me to pose the similar question for testing. I have found this to be a surprisingly difficult and complex set of issues to come to grips with. So, my ulterior motive in agreeing to chair this session was to bring some of the diverse expertise of this very capable group of people to bear on these difficult issues. I deeply regret that I was unable to participate in the discussion of these important issues.

I especially want to thank Mark Cornwell for the notes and slides that he prepared for the workshop wrap-up session on Friday. The summary that is given below is primarily my interpretation and some slight elaboration of Mark's slides.

1. Major Issues

The broad issue that was addressed by the session was the relative cost-effectiveness of testing and proving. To provide some structure for the discussion, several major questions were posed and discussed.

- Which approach is more cost effective in the long run?
- What are the components of cost?
- What determines cost effectiveness?
- How can it be assessed?

These questions led naturally to an attempt to formulate some working definitions. What is "testing?" What is "proving?" No definitive answers were

provided because the boundaries between the two are not sharply defined. Generally, testing was characterized as "back-end work" that is done after a product is produced, whereas proving was characterized as "front-end work" which is done before a product is produced. It generally was agreed that there is a wide spectrum of rigor that can be applied in both testing and proving.

2. Cost-Benefit Analysis

The session attempted to take a cost-benefit approach to comparing testing and proving. Several kinds of cost components were enumerated. The idea was that it might be possible to estimate the cost of each component for both testing and proving, to enumerate the various benefits of these components, and to estimate their value. The intent was that such an analysis of components might provide at least one consistent basis for comparing testing and proving. As the following sections show, the group got started toward this goal, but fell well short of completion.

2.1. Cost Components

The following components of software cost were identified. In most cases, they apply both to testing and to proving.

- Development of formal or informal specifications.
- Run time to perform testing and proving.
- Data reduction.
- Evaluation of test results.
- Detecting errors.
- Fixing errors. There have been some studies that indicate that the cost to fix an error roughly doubles as it goes undetected at each successive stage of software development. The following set of numbers are typical of the kinds of cost increases frequently quoted [Stahl 85]:

Requirements	$349
Design	$876
Code & Unit Test	$1750
Test & Integration	$12782

The way that these numbers are interpreted is that if a requirements error is detected during the requirements phase of a project, it costs about $349 to fix it, whereas if that same error is not detected until test and integration time, the cost is over $12,000. These numbers frequently are used to support the argument that early error detection reduces costs (and, therefore, you should hire an IV&V firm to save you money)!

- Development of support tools such as the following:
  * Simulators to provide test data.
  * Test analysis support tools.
  * Mechanical proof support tools.

Most of these cost components have both a human labor cost and also a cost of computing resources. The price of the human labor may vary considerably depending on the capabilities required for the various tasks.

2.2. Benefit Components

The following components of benefits of proving and testing also were identified.

- Increase in Mean Time Between Failures (MTBF).
- Reduced cost of errors to client.
- Political benefit to producer of no errors.
- Reduction in risk to client.
- Reduction in maintenance costs.

3. State of the Art

Some discussion was devoted to exploring the state of the art in testing and proving.

3.1. Today

At present, it is often programmatic issues that drive choices of testing or verification or proof methodologies.

Sometimes, the use of certain methodologies are mandated. (As one member of the session commented to me on Friday, "If the government mandates a methodology, it's cost-effective!") The A1 level of certifying secure operating systems is a case in point. These certifications mandate the use of formal top level specifications and proofs about those specifications.

Often, the choices are driven by management considerations. Assurance is a subjective thing, in that often what kind of assurance is provided depends very much on who it is being provided to. Unfortunately, the management that needs to be assured often is not well informed about various assurance techniques and, therefore, is not able to interpret the assurance evidence properly. What probably is needed is a variety of worked out examples of various projects that applying testing and proving techniques. These examples would enable managers and practitioners alike to see the effectiveness of various methods on particular examples.

The only advice that the group has to offer in this area is that, in making these decisions, it is necessary to weigh factors other than mathematical verification or testing. It is important to consider the interaction of various techniques and to define and approach assurance as early as possible in the project.

3.2. Tomorrow

Several trends were identified that the group believed will affect the cost effectiveness of both testing and verification.

- Narrowing the gap between formal specification languages and programming languages.
- Evolution of declarative languages.
- Machine architectures to support declarative languages.
- Networks, multiple processor architectures that support continuous testing.

- Possible widespread acceptance of a standard formal specification language.

4. Cost-Effectiveness Factors

The following issues were identified as affecting the cost effectiveness of system assurance.

- Criticality of properties assured.
- Isolation of properties in a small amount of code or spec.
- Formalizability of properties.
- Size of the system.
- Usefulness of a test result.
- Provability of a property.

It was observed that cost effectiveness is not a question that can be answered in isolation. It depends greatly on the characteristics of a specific problem.

5. Recommendation

As a final recommendation, the participants in the session suggested a handbook of various assurance techniques. This handbook would include the following items:

- Definitions of terminology.
- Enumeration of techniques.
- What kind of assurance they give.
- What benefits they provide.
- What they cost.

References to example efforts.

Such a handbook could provide a common ground for both the technical and the customer community.

Reference

[Stahl 85] Stanley H. Stahl, Sylvana Garlepp Nomicos. Cost-Effectiveness of Software Independent Verification and Validation. Technical Report ATD 85167, Logicon, September, 1985.

SOFTWARE PROVING AND TESTING

Harlan D. Mills
IBM Corporation and University of Maryland

This Workshop has produced many interesting and useful ideas about software proving and testing. In fact, I believe it is particularly important to consider proving and testing together. It is not a question of proving or testing. Neither is sufficient. But the combination of proving and testing is the best idea on the software horizon today.

Testing has been a fixture in software from its inception. Dijkstra has pointed out that testing shows the presence of errors but not their absence. But statistical testing from a population of software usage scenarios can do better, by providing scientific, objective evidence of software reliability.

Proving is of more recent origin. Dijkstra's motivation in structured programming was to reduce the lengths of proofs. Variable free proof procedures that scale up to large programs, but with increased individual fallibility, can be used with additional checks and balances to decrease team fallibility to the point of eliminating the need for software debugging prior to system testing.

Formality is critical for both proving and testing. But formality should be defined at the semantic level, not the syntax level, with the goal of repeatability in human interpretation rather than this or that representation. For example, a formal specification for a data abstraction should be defined by a mathematical relation, however represented, plus a probability measure on the iterated product sets of its domain, to define usage scenarios.

The combination of software proving and testing allows the definition of software development under statistical quality control. The short history of software would seem to indicate that software development, a creative process, and statistical quality control, an objective process, are contradictions in terms. However, with formal specifications that include probability measures on usage scenarios, the statistical quality control process can be imposed literally on software development.