Directions for U.S. Research and Development Efforts on Software Testing and Analysis

A Report from the Workshop on Directions in Software Testing and Analysis

San Diego, 1-3 August 1989
(Revised December 1990 & September 1991)

Leon Osterweil
Department of Information and Computer Science
University of California, [1]

Lori A. Clarke
Department of Computer and Information Science
University of Massachusetts, Amherst

[1]This material is based on work supported by the Office of NavalResearch and Office of Naval Technology under Grant No. N00014-89-J-3049

Abstract

This report sets forth a national research agenda aimed at improving the *quality* of computer software–an invisible, yet increasingly crucial, element of our national infrastructure. The report is a summary of a meeting of research and industrial leaders, held at San Diego, California, 1-3 August 1989, under the sponsorship of the U.S. Office of Naval Research and Office of Naval Technology. This workshop identified nine promising directions of analysis and testing research, aimed at strengthening the underlying foundations and furthering the exploration of innovative and novel ideas. Most of these research directions show promise of producing technology transferable into effective use in industry within the decade.

1 Software Testing and Analysis in the United States Today

1.1 Pervasiveness of Software

Computer systems are rapidly becoming an integral part of nearly every facet of our economy and life. Government and industry rely on computer systems to assemble, process, store, and disseminate data. Neither could run their day to day business without computers. Individuals use computers in their daily lives to do such tasks as personal banking and to control, often invisibly, many of the common household appliances upon which they rely. The military uses computers to help in assuring national security. Airlines rely upon computers to control aircraft in flight as well as to handle logistics on the ground. The medical profession uses computers in life sustaining medical equipment, such as pacemakers, as well as in support of more mundane tasks such as dispensing medication and planning menues.

As the importance and pervasiveness of computer systems have grown so has the implicit need for them to perform reliably. Computer system failures are becoming an increasingly familiar part of our lives. While some are benign, or even amusing, some are profoundly disconcerting. Automatic tellers spewing forth money to undeserving, yet gleeful, customers are reported in the news with amusement. The computer error that required a major New York bank to borrow tens of billions of dollars overnight, however, was far less amusing. This error cost the bank millions of dollars and threatened to destabilize our national financial system. Computer failures have also figured in crashes of airplanes and trains. A computer simulation inadequacy currently threatens the Trident II missile program. Computer failures caused the postponement of more than one Space Shuttle launch and nearly caused the failure of the first manned landing on the moon. Recently, a computer failure caused a nationwide long-distance telephone outage. The failures in the examples cited here were caused, not by computer hardware, but by computer software.

It is important to recognize that, even though computer software is invisible and intangible, it is nevertheless now one of the most important elements of our national infrastructure. Roads, bridges, railways, schools, and banks are quite visible, but computer software, while invisible, is nevertheless used in significant ways in the operations of railways, schools, and banks, and is used critically in the design and construction of roads and bridges. The same concern for quality in constructing and maintaining these more tangible forms of infrastructure must also be shown in developing and maintaining our computer software infrastructure.

Although computer software has been catapulted into this position of national importance, it is unfortunately not possible to guarantee that it deserves the trust and reliance currently placed upon it. Indeed the previous enumeration suggests that it is not sufficiently trustworthy. Heroic human effort has made our critical software systems workable, but the technology to convey needed assurances about these systems does not exist and will not be developed in the foreseeable future, unless dramatic steps are taken.

1.2 Computer Software in Major Initiatives and Grand Challenges

Much attention has been focused on new national initiatives such as the Strategic Defense Initiative (SDI) and the NASA Space Station. In addition, there has been considerable discussion of such scientific "Grand Challenges" as the Genome project, the Superconductor Super Collider, and voyages by humans to the planets. The success of all of these initiatives will depend upon massive amounts of trusted and reliable software systems. The challenge of developing such software systems has received some attention. The challenge of being able to assure ourselves that these systems can be trusted to operate acceptably well has received even less attention.

It was the consensus of workshop participants that computer software must be treated as a "Grand Enabler" in our attempts at these initiatives and challenges. It was also argued that our lack of effective software analysis and testing technology has caused our national infrastructure to be more fragile and untrustworthy than is commonly assumed. This threatens to become a dangerous chokepoint in our push towards our national aspirations.

1.3 Software Testing and Analysis–The State of the Art and State of the Practice

1.3.1 The State of the Practice

Computer systems in use today appear to work rather well. While computer failures of varying degrees of severity are reported regularly, a casual examination of the situation does not indicate the existence of a critical problem. Deeper examination is more troubling.

Statistics gathered over the past ten or fifteen years show that testing, analysis, and debugging costs usually consume over 50% of the costs associated with the development of large software systems. This cost is too high, especially in light of the results achieved. Examination of these activities indicates that most of the work is done manually and in an ad hoc fashion. Humans are relied upon to use their intuition and judgment to create testcases, evaluate the results, and decide when sufficient testing has been done. Nontrivial automated support for these activities is the exception rather than the rule. Further, there are no widely accepted formal guidelines or standards for determining when adequate testing and analysis have been done. It is rare that software systems are delivered with any assurances other than that a great deal of effort and resources have gone into testing.

It is expected that new software systems will fail, perhaps frequently, in the early days and weeks after delivery. Customers and software contractors usually work together to identify and correct these errors. Usually the result is increased reliability and effectiveness. Usually systems are accepted after system performance has been stabilized to the satisfaction of the customer. Many systems, however, never satisfy their customers. Some are abandoned forever, while some must be rebuilt from scratch. In all cases, the process of testing, evaluating, correcting, and deciding to deploy software systems is one that rests

almost exclusively on unsupported, sometimes haphazard, human labor and judgment. Our successes to date are largely tributes to the efforts and dedication of humans. While the research community has made significant contributions to testing and analysis practice (see next section), the need for more effective testing and analysis technology, resting upon firm theoretical foundations, is acute. Tools to help in the analysis and testing of software could significantly reduce human labor, and accurate, acknowledged measures of test adequacy and software reliability could reduce our vulnerabilities due to reliance exclusively upon (fallible) human judgment.

This software reliability crisis is generally recognized in software development circles. Large software systems require large testing activities. As systems become larger and more complex, they tax the intuition and judgment of humans in deciding how to test. In addition, new hardware technologies such as parallel computers and distributed networks entice software developers to create more complex and non-intuitive software systems that dangerously tax the effectiveness of testing personnel. Likewise, new paradigm software, such as logic programs and neural nets, raise the need for new testing and analysis techniques. Some software system developers have sidestepped the more modern technologies at least partly because of risks that systems built upon them might not be adequately testable. This indicates one way that an underdeveloped analysis and testing technology can choke off progress on aggressive initiatives.

1.3.2 The State of the Art

It can be argued that research contributions to software analysis and testing have not yet had a major impact on industrial practice. It is clear, however, that significant research results have been obtained, some impact on practice is evident, and there is clear potential for further research progress and more effective technology transfer.

Among the most significant ideas that have been developed by the research community are:

- the use of graph theoretic models of software control and data flow as the basis for measuring testing coverage. At least 20 years ago researchers had suggested that combinatorial structures such as graphs could be used to model computer programs effectively. A program flowgraph was defined to be a graph whose nodes correspond to atomic execution units such as statements, and whose edges correspond to possible transfers of control between such units. Using this model, program loops correspond to flowgraph cycles and program execution sequences correspond to paths through the flowgraph. Using this formalism, testing adequacy criteria were initially quantified as the percentages of nodes, edges, and loops exercised by testcase execution. These criteria have been extended to include more complex data and control dependencies. Such graph-based testing criteria are generally regarded as being cost-effective but do not provide the high-level of reliability required for safety critical applications.

- the use of static analyzers for efficiently demonstrating the absence of certain kinds of errors. Using flowgraphs, values used in computations can be traced back to the statements whose executions created them, and conversely values created by the execution of a statement can be traced to the sites at which they are used. Often irregularities and errors in patterns of such creation and usage can be detected. Similarly, user-defined patterns of potentially unsafe sequences of events can also be detected. Because the flowgraph provides a model of all possible executions of the program, it is possible for these checks to be exhaustive, and for them to therefore indicate that certain types of errors cannot occur in any program execution. Some modern compilers now carry out some of these analyses, many of which were first suggested by testing and analysis researchers.

- the use of assertions to capture intent and to serve as the basis for 1) statically analyzing and dynamically monitoring the adherence of a program to this intent, and 2) reasoning about the program itself. All testing and analysis activities ultimately boil down to attempts to demonstate that the program either does or does not behave as intended. Thus a firm specification of intent is essential to effective testing and analysis. Researchers established this over 20 years ago, and proposed mathematical formalisms, such as predicate logic, as vehicles for expressing intent rigorously. Such rigorous specifications of intent are usually referred to as assertions, and are used to specify the conditions that are expected to hold at the beginning and end of execution of a section of code, as well as at key points in between. Researchers have shown that these assertions can be used to state and prove theorems about whether or not program executions must always behave as intended. They have also devised systems for using such assertions to monitor actual program executions to determine if and when they begin to stray from intended behavior. Awareness of the importance and utility of assertions of intent has spread to the practitioner community.

- the design of comprehensive environments for effective integration of tools in support of testing and analysis. Early analysis and testing tools developed by researchers proved to be large, and yet often narrow in focus and efficacy. This led to early interest in developing large suites of tools to support a broader range of analysis and testing activities, and to the underlying issues in software engineering environments. Software environment and tool integration research continues to be motivated, at least in part, by the need to support effective integration of testing and analysis tools with each other and into larger software development and maintenance processes. This research is finding its way into practical application in the form of Computer Aided Software Engineering (CASE) tool integration systems and frameworks.

- the application of software fault models to error detection. By incrementally building on their previous results, the testing and analysis research community has evolved a

model of fault detection that rigorously describes the necessary and sufficient conditions for a fault in the code to be revealed as erroneous output. It is precisely because erroneous intermediate values are masked out by subsequent computations that ad hoc testing is ineffective. The software fault models describe conditions that must be satisfied for a fault to be revealed at the originating statement as well as at subsequent statements in the software. Although too costly to be implemented in their entirety, these models provide the basis for automating selected aspects of the model. Successful prototypes that do just that are currently being experimentally evaluated as potential CASE tools.

- the development of more rigorous requirements and specification formalisms – especially executable specification languages. The growth in visibility and perceived importance of requirements and specifications over the past 20 or more years has been fueled by a variety of concerns. The need to carry out effective analysis and testing is one such key concern. Researchers have successfully argued that testing and analysis are exercises in comparing product software and its behavior to specifications of intent. Testing and analysis can be no more rigorous and definitive than are the specifications used as bases for comparison. Accordingly significant effort has gone into fashioning formalisms that suffice to support definitive analysis and testing. This research has led to the creation of increasingly rigorous formalisms. The critical importance of such rigor to effective analysis and testing has been a key factor in increasing their appeal and acceptability to the practitioner community. Executable specifications are one particularly appropriate example of such research.

Some of these ideas have clearly had favorable impacts upon practice. For instance, graph theoretic terminology is commonly used by practitioners in discussions about testing adequacy. In addition the use of mathematical logic to capture the intent of software systems has been used successfully to increase the trustworthiness of safety critical software components.

It should be expected that the research community's lead in developing and integrating large and diverse suites of testing and analysis tools will be taken up by practitioners in coming years. Primitive tools, such as flowgraph and callgraph displays and testpath generators, have been developed and put into practice. More advanced tools, such as dynamic assertion checkers and static dataflow analyzers, are on the verge of introduction into the commercial marketplace and will see industrial use in the near future. Integrated suites of such tools will begin to appear shortly as they are commercialized by the CASE industry.

1.4 Current Obstacles and Problems

Clearly it is highly desirable to accelerate the process of developing new and more effective analysis and testing technologies in research laboratories and rapidly transferring them to

industrial use. Unfortunately there are many serious obstacles to this.

For example, there are fundamental theoretical obstacles to the development of some of the most important testing and analysis techniques. Early researchers explored the obstacles posed by such barriers as the incompleteness of first order logic and the NP-Completeness of many key graph-analysis algorithms. They showed that automatic tools cannot always be counted upon to generate test data, force execution of certain program paths, and predetermine patterns of concurrency in a program. As a result, some of our most important analyses (such as automatic determination of the executability of a given program path) are impossible or impractical for many programs.

The recognition of these obstacles has obliged us to search for effective heuristics and special case procedures, rather than the general solutions that would be more desirable. Here too, however, there are serious problems posed by difficulties in definitively evaluating these heuristics. The lack of underlying theories in support of testing and analysis thwarts such evaluations.

Once successful analysis and testing tools have been developed by researchers, there are other problems. These researchers usually evaluate their ideas through the construction of prototypes that concentrate on implementation and evaluation of the new ideas, often by ignoring (temporarily) the problems posed by troublesome language features (such as pointer variables). Contemporary programming languages almost universally contain such troublesome features. For example, most common uses of pointer variables (e.g., in C) severely complicate the work of static analysis tools. In addition, most research prototypes are not designed to handle large programs. Researchers concentrate on demonstrating the validity and importance of their research idea first. In doing so, they often create tools that are too slow and too restrictive to be applied to the large programs produced by industry.

The result is that most research tools are not capable of dealing with programs as large and complex as those commonly found in actual practice. To evaluate such a tool fully it is usually necessary to construct a larger, more ambitious prototype–usually at great cost. To successfully transform such large prototypes into industrial strength tools requires considerable additional expense. Thus, the cost of evaluating promising research ideas in industrial contexts can be quite high.

There are other obstacles that are organizational. For example, testing and other software quality assurance activities are generally carried out at the end of the development cycle. Because these activities are done at the end of the development process, major errors that might have been readily detected if analyzed earlier are often not detected until the end of the process, when their remediation is expensive. It seems clear that it would be most cost effective to carry out testing and analysis continuously throughout the software development lifecycle. Unfortunately, there is currently little technology for supporting this. The development of this technology is seriously hampered by lack of general agreement on rigorously defined requirements and design notations, with the sorts of non-trivial semantics that could be the basis for useful analysis.

As a result, software testing and analysis generally remain relegated to the end of the development process when correction tends to be most costly. Often they are further hampered by inadequate planning. Software quality assurance organizations often find that the time and resources allocated for testing and analysis are insufficient. To make matters worse, they are often cut short because of schedule slippages and cost overruns in earlier phases. Finally, increasingly we find that, at least in part due to the problems and inattention just described, the testing and analysis activity is carried out by the most junior and least competent personnel in an organization. The result is often a poor job.

Unfortunately, our difficulties seem to be getting more severe rather than more tractable. The software systems we are attempting to build are increasingly large and complex. Newer systems are often distributed, concurrent and/or real-time, yet there is very little testing and analysis technology for handling such software. These new software systems are being embedded in increasingly critical applications, increasing the importance of being sure of their quality and safety. New languages and language paradigms are being introduced. Some of them contain linguistic constructs whose analysis poses new and challenging problems. Software costs continue to rise, making the cost of constructing industrial strength versions of research prototypes ever higher. Labor costs continue to increase, yet testing and analysis remain labor intensive.

1.5 Timeliness of an Initiative

While the United States has been slow to recognize the dire need for improved support for software quality, the Japanese and European Community have launched major efforts focused at improving software quality. In Japan, most major universities have faculty involved in software engineering research, addressing some aspect of the problem of developing reliable, complex software systems. Japanese industry is usually the first to experiment with software testing and analysis research results, investing the resources to develop robust prototypes applicable to industrial software. Unfortunately, results from such experimentation are generally proprietary and do not impact US industries. There is evidence however that such experimentation is changing the ways in which Japanese industries produce software. There is a concern that Japan will become the recognized leader in quality software production.

The European Community has engaged in several wide-scale efforts to address the software quality problem. The Esprit, Eureka, and Alvey projects brought industrial and academic researchers together to explore various approaches to improving software quality. One noticeable outgrowth of these large, well-funded efforts has been an increased interest in investigating formal methods for supporting software development activities. Europe is currently seen as the world leader in developing and applying formal methods to the software development process.

The realization that software testing and analysis problems are severe and growing

has not been lost on the US testing and analysis research community. Unfortunately, the magnitude of the problems seem to be growing faster than the community's ability to respond. New basic research must address the problems being posed by new classes of software. Additional research is needed to make further inroads into the more classical problems. Intensified effort is needed to promote rapid and successful transfer of effective research ideas and systems into practice.

To do this, the research community is identifying the key problems at hand, and taking steps to agree upon an agenda for attacking these problems. Clearly it will be necessary to intensify its level of effort. This will require sharply increased funding, a widening of the research community, more collaboration with industry on projects aimed at evaluating research ideas and tools, and continuing work on planning and prioritizing future efforts.

This report describes the steps that are being taken to develop a research agenda and to bridge the current gap between software testing and analysis researchers and industrial practitioners. It proposes a plan for raising current levels of effort and support to mitigate the danger that insufficient software quality will become a chokepoint in efforts to realize our national aspirations.

2 Research Directions

The workshop participants strongly agreed that there needs to be both a strengthening of the underlying foundation of testing and analysis research, as well as exploration of highly innovative approaches. The first type of research is urgently needed because the work done to date has not been adequately organized into a coherent body of knowledge, based upon a solid formal foundation. As a result, it is difficult to know what capabilities are at our disposal or how to evaluate or build new and innovative capabilities upon this knowledge. The second type of research is urgently needed because the explosive growth in the development of larger, more diverse, and more critical software systems is outpacing our ability to devise effective analysis and testing capabilities for them.

To pursue these two types of research, the workshop participants identified nine specific areas of inquiry as being most likely to yield important results over the coming years. It is important to realize that these nine research thrusts are not necessarily independent. A particular research project might contribute to several of these areas simultaneously. Because of this interaction, the participants did not prioritize the areas. Progress in any one of these areas has the potential to improve software quality significantly. Moreover, a breakthrough in one area is likely to impact other areas, providing new insight that those areas can build upon as well.

In this section we itemize the nine areas and outline some of the research directions that might be pursued in each.

2.1 Theoretical Foundations

While progress has been made on developing a theoretical basis for testing and analysis, it was widely agreed that more research is needed in this area. Formal models have been developed for data and control flow, fault propagation, test criteria adequacy, and verification, among others. These models have provided valuable insight into the problems we are addressing and have guided the development and evaluation of many of the existing techniques and tools.

Still there is a sense that the theoretical models developed so far have only scratched the surface. Greater leverage and understanding will be gained by accelerated research into new, more comprehensive models upon which to base future analysis techniques. Better models are needed to support statistical or probabilistic testing models, fault detection, and safety analysis, to name just a few. In addition, models must be developed to deal with more complex computing domains. In particular, formal models must be developed that include real-time constraints, concurrency, and distribution requirements. In some cases, existing models can be extended to deal with these new dimensions but in many cases new theories and models must be explored. Without such a foundation to build upon, future progress will be greatly hampered.

2.2 Analytic and Empirical Evaluation

Ideally software testing and analysis research results should be evaluated according to some agreed upon objective standards. Such standards could be used to render new research results comparable to previous results, giving researchers an indication of those ideas most worth building upon and practitioners an indication of which technology to apply. Unfortunately such standards do not exist. New techniques are proposed and urged for consideration based upon limited experience and evaluation.

Consequently, attempts to supply cogent evidence of a new technique's advantages are seriously hampered by the lack of agreed upon standards for evaluation and the lack of support for carrying out extensive evaluation. Thus, the workshop participants strongly urged support for both analytic and experimental evaluation of testing and analysis techniques.

Rigorous analytic evaluation can provide absolute comparisons among testing and analysis techniques. Such comparisons provide worst case or average expected results and therefore provide a baseline upon which to measure experimental results. Such rigorous analysis requires a common formal model upon which to evaluate a related class of techniques and thus is an impetus for further work on theoretical models.

Unfortunately it is often difficult to build such a model. Moreover, even when analytic results are available, experimentation with operational software is still desirable since analytic comparisons all too often evaluate worst-case behavior for limited classes of programs (e.g., well-structured, no dynamic data types). Experimental studies attempt to approximate the typical operational input to a technique by carefully selecting a small but

representative set of programs. Observing the actual performance of a technique on such software allows one to extropolate about the actual expected behavior of the technique when applied in an industrial setting.

Two kinds of experimental studies have proven to be effective in the past. One is smaller in scale, but especially designed to be repeatable by other researchers. In such cases, successful repetition forms the basis for scientific confidence and consensus building. Because of the nature of software development, another kind of experimental approach, involving the careful in-depth study of the application of a technique or class of techniques to a large, complex system, is also of considerable value. Although usually too costly to be repeatable, such experimental case studies provide valuable feedback about actual operational behavior and about the potential scalability of the techniques being examined.

Both of these types of experimental studies have been hampered by the unavailability of a repository consisting of suites of realistic programs with known faults, software testing and quality assurance processes, and benchmark testing and analysis tools. The development of this repository is itself a research activity. Broad acceptance of it can only evolve slowly. It will require an iterative process during which the repository's components are constantly improved and adapted, with community confidence continually growing as these components are used in a variety of evaluations. Carefully designed pilot projects should be quite useful in helping with the development of such a repository.

2.3 Large-Scale Systems

It was agreed that software testing and analysis research results fail most dramatically when they are attempted for large software systems. Two major obstacles to effective testing and analysis of large-scale software systems were identified. One is that current techniques have focused primarily on programming-in-the-small concerns. New techniques must be developed to address the testing and analysis of large software systems. The second is our failure to produce prototypes that can be used, even experimentally, on industrial size systems written in the most commonly used languages.

It was generally agreed that while a variety of testing and analysis tools and techniques exist, the majority focus on testing or analyzing a single module in isolation. Little research has been directed at testing and analyzing large software systems. Such systems may be composed of thousands of modules communicating through a number of complex interactions. Special techniques must be developed to assess the quality of these systems during integration testing, system testing, and regression testing. Currently, we do not know which of the existing techniques can be successfully extended to support the analysis of large-scale systems. Certainly, with appropriate extensions, some of the existing techniques could be modified to address the testing and analysis needs of large-scale software systems, and research into these extensions should be encouraged. Most importantly, investigations into new techniques specifically designed to deal with large, complex systems must be pursued.

A major drawback to investigating analysis and testing techniques for large-scale systems is our inability to evaluate techniques on industrial size software systems. As noted, the majority of testing and analysis tools have been developed by researchers as small prototypes. These prototypes have been devised to explore innovative ideas and to demonstrate their potential. Generally they are not designed to be efficient or easy to use. Consequently they are unacceptably slow and cumbersome when applied to the large software systems produced by industry. Often they are designed to analyze programs written in only a subset of a particular language. As a result, these prototype tools are often totally useless for analyzing and testing real programs.

The net result of this situation is that industry derives little or no direct benefit from the large number of potentially valuable research prototypes. Combined with the lack of expermental studies (described above), this leaves industrial practitioners without a reliable basis to use in deciding which (if any) research tools and techniques might be of value.

It is necessary to embark upon an effort to develop industrial strength versions of the more promising testing and analysis techniques. In many cases this will entail the complete redevelopment of existing tools to make them acceptably fast, robust, easy to use, and complete. These activities will require considerable tool development effort, but will lay the foundation for experimental evaluation.

The development of industrial strength versions of current research prototypes will also lead to important new research. As already noted, many research prototypes do not handle certain language features because they are difficult to analyze. Industrial strength versions of these prototypes will have to address these problematic language features.

2.4 New Software Paradigms

Innovative research is required to support the effective analysis and testing of the rapidly widening variety of software systems that are now being developed. These new systems may employ parallel and distributed hardware, as well as new paradigm or multiparadigm languages. Effective testing and analysis of such software systems is currently impossible due to the lack of suitable tools and techniques.

In view of the fact that the most critical (especially life-critical) applications necessitate real-time programming, it is shocking to see that so little research into testing and analysis of such software has been embarked upon. Likewise, there has been very little research into the testing and analysis of distributed and parallel systems. Distributed and parallel computation are looked upon as vehicles for effecting the application of greater computing power to a problem through either large- or small-grained parallelism. Thus it is anticipated that more and more systems will employ distributed or parallel computation to meet their ever growing need for more computing cycles. The lack of effective means to test and analyze such systems must be viewed with concern.

Virtually all testing and analysis research and prototype tool development has addressed

software written in algorithmic languages. There has been, however, steadily increasing interest in logic programming, functional programming, and neural-net programming. Some proponents of these new linguistic paradigms assert that their use implicitly assures higher quality software. There is insufficient research to either support or refute such claims. Clearly research aimed at the analysis and testing of software written in these new languages is needed. It is also clear that the effective testing and analysis of such software will entail the devising of innovative techniques.

2.5 Massive Computing Power

The research community has long understood that effective software testing and analysis can consume enormous quantities of computing resources. In the past, many promising testing and analysis techniques had to be rejected because it was assumed that the computing power needed to make then effective was simply not available and would not be available in the foreseeable future. The sorts of massive computing power that are required are now becoming more readily available. For example, desktop workstations often contain processors having speeds in the tens of MIPS and memories measured in the tens of Megabytes. These workstations generally reside on networks that may contain dozens of similar workstations, many of which may sit idle for extended periods of time. These networks can also be increasingly expected to contain massively parallel computers, largely underutilized backend co-processors and servers with multi-gigabytes of rapidly accessible storage. Efforts to exploit this computing power effectively should lead to research into the redesign and alternative implementations of known techniques. This can be expected to open the door to totally new and more effective approaches to testing and analysis.

For example, research prototype testing and analysis tools based upon source code instrumentation often multiply the size of the subject program source text by as much as a factor of ten. Execution of such instrumented programs may require similar multiples of time. Such bloated programs have, until recently, been considered to be impractical to execute. As a result most of these research prototypes have never been evaluated and applied in industrial contexts. The recent availability of massive computing power raises the possibility that such tools might now receive this sort of industrial evaluation.

Likewise, static analyzers have required enormous amounts of storage space and execution time. The annotations required for static dataflow analysis of sequential programs may increase the size of the corresponding object code by several multiples. The enormous amounts of time required to complete the analysis of these large representations have also blocked their application. In addition, techniques such as symbolic execution have remained in the research laboratory because of their known voracious appetites for computing resources. The need for expensive constraint solvers and reasoning engines has also been an obstacle. Similarly, large amalgams of complex tools have been proposed, but not implemented, because of fears that they would consume excessive quantities of computing

resources.

It would be a mistake to underestimate the difficulty of redesigning and reimplementing testing and analysis techniques based upon these new and powerful hardware platforms. To make the most of these machines and their different architectures, many tools will have to be parallelized and distributed. It is expected that the need to do this should lead to deeper understandings of these existing techniques, which might then lead to qualitative improvements. This has indeed been the experience in other fields within Computer Science. Of course, massive parallelism is no panacea. Those techniques whose needs for computing resources increase exponentially with the size of subject programs, will remain impractical even on foreseeable hardware configurations until and unless suitable heuristics are devised. The availability of such massive computing resources, however, might suggest entirely new heuristics and approaches to software testing and analysis that would lead to new breakthroughs in the field.

2.6 Process-related issues

Research needs to be focused on modeling software products and the processes by which they are developed. Unfortunately, software development is currently guided by informal procedures, coordinated through ad hoc verbal communication. It is not surprising that this often results in inferior software products. Clearly this situation is unacceptable. Instead, software development should be a systematic process. It should be sufficiently visible to render it susceptible to effective management and to assure a high-quality product.

There is a growing consensus among software engineers that software development products and processes must be modeled and defined clearly, completely, and rigorously, thereby rendering them more visible, understandable, and analyzable. For example, once software development processes are modeled in this way, it will be possible to indicate precisely how and when testing and analysis activities are to be carried out and to evaluate and improve these processes. Based on these models, it will be possible to determine when earlier phases of the development process are beginning to consume resources needed for subsequent testing and analysis. Perhaps most important, it will be possible to enforce the specified testing and analysis activities in those earlier phases, where they are known to be most effective.

This line of research can also serve to guide the effective integration of testing and analysis tools. Software process models can provide insight into how tools implementing various techniques can be integrated, exposing duplication, inconsistencies, and the need for new tools and techniques that might not yet exist. Such models might also serve to guide the integration of these tools into canonical testing and analysis processes. Models of these processes themselves might then become objects of studies to establish characteristics and patterns of efficacy and weakness. Carefully investigated analysis and testing process models might then be applied more routinely and with increasing certainty and confidence in the accuracy of results.

Software process models can also be used to indicate how testing and analysis can be integrated with earlier lifecycle phases such as requirements and design. Such models should also address the issue of how errors discovered during testing or analysis can necessitate changes to requirements and design and trigger the need for focused redevelopment and retesting.

The software testing and analysis community has a further role to play in assuring the quality of these software process models. If process models are to be effective in directing industrial software development, it is important to have assurances that they are themselves of high quality and have demonstrated benefits. Thus the testing and analysis community must embark upon research to develop full lifecycle testing and analysis processes and to evaluate these processes to assure their effectiveness.

2.7 Application-Specific Domains

Often narrowing the scope of applicability of a technique enables one to devise analyses that are more powerful over the restricted scope than those that are applicable over an entire wider scope. For example, it has been noted that many research prototypes support testing and analysis of programs written in only a subset of their subject languages. While this restricts their applicability, it enables the tools and techniques to bring more powerful analyses to bear. Researchers have emphasized that the added power of their tools and techniques might well be worth the price of accepting restrictions on the way in which software is developed. This premise seems plausible and should be explored as a research activity.

To explore this hypothesis it is necessary to experiment with restricting the domains to which testing and analysis tools and techniques are to be applied, and measuring the effectiveness of the tools and techniques on software within those domains. One example of this sort of research would be to specify a dialect of some well-known language and demonstrate how a tool or technique of agreed upon value might be made more effective by coding within the specified dialect. Another similar line of research might be to study the effectiveness of tools and techniques designed for software within a certain given application domain. It is reasonable to expect that software in specific application areas (e.g., engine control, payroll processing, or message switching) might have specific characteristics that could be exploited to create more powerful testing and analysis tools for such software. Careful identification of these characteristics would allow researchers to recognize application domains with similar characteristics that could also exploit the increased power of these tools and techniques, thereby widening the scope of their applicability. Still another research direction might involve specifying a particular development process (perhaps including specific requirements and design techniques) and then determining the value of the additional analytic power gained by following that process.

This line of research should lead to more understanding of how to develop software

for increased testability. Currently most research is directed towards tools that are broadly applicable. A promising alternative to this line of research would be to study the cost/benefit tradeoffs of tools and techniques with narrower applicability but greater power.

2.8 Requirements and Specification

Research into requirements and specification mechanisms should be sharply accelerated, with particular focus on how to make them more effective in supporting testing and analysis. All testing and analysis entails the comparison of software products (e.g., code, designs) against specifications of what they are supposed to do. Insofar as the specifications are vague and imprecise, testing and analysis will be correspondingly indecisive. Firm, rigorous specifications of the effects that software systems are intended to achieve are essential if testing and analysis are to offer sufficiently definitive assurances.

First order logic has been used effectively as a formalism for specifying the functional intent of software systems. It has been effective as the basis for rigorous verification of the functionality of both code and designs, and for effective dynamic testing. More recently temporal logic has been investigated, particularly as a specification formalism that is useful in verification of real-time software. Other formalisms such as constructive type theory have also been suggested and are in need of further research and evaluation.

It seems particularly important to accelerate research into formalisms and metrics capable of effectively supporting the capture and evaluation of non-functional requirements. For example, formalisms for specifying software safety, robustness, and modifiability are needed. With such new formalisms comes the need for corresponding tools, technologies, and metrics that can be used effectively to compare software products (e.g., code, designs) to specifications of intent created in these formalisms.

2.9 Software Evolution

The development of tools and techniques to support retesting and reanalysis during the evolution and maintenance of software systems was also identified as a critically important activity, ripe for exploration. It is often observed that "maintenance" costs over the lifetime of a software product generally exceed initial development costs by a factor of two or more. The term maintenance covers a broad spectrum of activities, including bug fixing, streamlining, and capability enhancement. All of these activities require alteration to a basic software product, and hence all of them require retesting and reanalysis to assure that the altered product satisfies new or changed requirements, while continuing to meet old or unchanged requirements.

Existing testing and analysis tools and techniques generally operate under the assumption that the software upon which they are to operate is totally unknown and unfamiliar. Thus they begin their work from a base of no prior knowledge of the subject software. In a maintenance situation, it should be expected that the subject software has been tested

and analyzed previously–perhaps many times–and that there should be a potentially large base of information about it. It is obviously desirable to find a way to exploit that information and reuse at least some of it to speed and enhance the retesting and reanalysis of the altered software. Some early tools already do this in rather limited ways. Intelligent editors do incremental parsing of code and some do limited amounts of incremental semantic analysis. Incremental compilers carry out still more incremental reanalysis. In addition, regression testing procedures can be viewed as software testing processes that are designed to do automatic (but usually non-selective) retesting in response to maintenance changes.

More such tools and techniques are needed. Especially in view of the fact that most software lifecycle costs are incurred during maintenance and so much of total maintenance cost is attributable to retesting and reanalysis, it seems only reasonable that much greater emphasis should be placed upon efforts to reduce these costs. Some existing tools designed for use during development can be redesigned. But what is really needed is the development of a discipline of "reanalysis and retesting" aimed at inexpensively certifying that changes could not have altered the validity of certain analyses that had been performed previously. This seems to be a new form of analysis, rooted in more traditional forms of analysis.

3 Plan for a Testing and Analysis Initiative

This report strongly recommends that a major initiative in software testing and analysis be embarked upon at the soonest possible time. Workshop participants had little difficulty identifying promising research directions that could and should be embarked upon. They recommended accelerating support for this research and for efforts to transfer it to practice effectively. They stressed that this initiative must not come at the expense of other basic and applied research in computer science. They also stressed that this initiative should strike a balance between increased funding for large group projects and increased funding for smaller, individual projects. The purpose of this initiative would be to provide the needed resources, an organizing framework for the work, and an impetus to broaden and intensify critical software testing and analysis research.

3.1 Research Thrusts

The research to be carried out under this initiative would fall into two broad categories– 1) research aimed at consolidating and strengthening the underlying foundation and 2) research aimed at higher-risk endeavors

Section 3 enumerated a number of research directions that addressed both consolidation and innovation and are expected to lead to important new results in software testing and analysis. We propose to focus significant resources on the vigorous pursuit of these research directions. As noted in the previous section, we propose to establish suitable evaluation frameworks that would be used to evaluate the results of this research and to foster

technology transition based on extensive evaluation.

3.2 Funding Characteristics

Research leading to the formulation of innovative ideas and approaches has been successful in the past and must be supported at increased levels in the future. In addition, there must be an increased level of support for evaluation. Often evaluation is best pursued through full-scale prototype experimentation, entailing conceptualization, and prototype system development, evaluation, and iteration. Clearly such an activity is resource intensive. In the past, due to lack of resources, testing and analysis research has been obliged to focus on conceptualization, followed by primitive prototyping activities and ineffective evaluation. It has been impossible to build effective prototypes that can serve as the basis for thorough experimentation and evaluation. This has made effective iteration impossible.

Thus, larger research projects, funded over longer time periods, are necessary. These larger projects should be collaborations among system developers, evaluation experts, and testing and analysis researchers. They must be supported at suitably high levels of funding and must be expected to last for at least 3-5 years.

Smaller research projects, emphasizing exploration of higher-risk approaches are also needed. More modest funding levels and shorter time frames would be appropriate for these projects.

3.3 Cost and Schedule Estimates

It is hard to predict which of the proposed research areas will prove to be the most ripe for exploration and which will be best left until groundwork is laid by others. It seems clear, however, that in the steady state each of the proposed research areas would have to be explored by no less than two major research groups. Thus we anticipate that approximately 20 large research projects would have to be funded. In view of the importance of careful evaluation of research results, it is recommended that most of these projects include teams of professionals. Our estimate is that such projects would require initial funding at the level of approximately $200,000 per year. We recommend that approximately ten smaller projects emphasizing higher-risk approaches also be funded at a level of approximately $100,000 per year. Thus, we estimate that the initiative would require funding of approximately $5 million annually in its first years. As the initiative progresses, we anticipate that it would broaden in scope and required funding.

As these projects will entail the development and evaluation of research prototypes, each should be expected to last for at least three years, and perhaps as many as five years. Projects of this length can be expected to be effective in creating important new technology. Surely we expect that prototype systems resulting from these projects would be made available for demonstration project evaluation within the first three years. Experiences

with these prototypes would be the basis for decisions about continued funding of the projects and continuation of the initiative itself.

4 Expected Results

While the projected cost of such an initiative is admittedly quite high, we believe that the cost of not embarking upon such a program is considerably higher. Computer software costs in the United States are measured in the tens of billions of dollars annually. Thus the cost of this initiative is considerably less than 0.1% of what we are currently paying for software.

In addition, the costs we are incurring due to poor quality of the software is no doubt even greater. Poor quality software leads to poor utilization of hardware, and costly errors and inefficiencies. Delays in fielding effective software often leads to late delivery and deployment of systems, tying up capital and slowing the progress of large industrial and government systems.

The initiative would increase the size and vitality of the software testing and analysis research community. The small community of current researchers produces a small number of new Ph.D's, and, as a result, the community stays small. There seems to be no quick and easy cure for this situation. A major research initiative should, however, attract researchers from related disciplines within Computer Science. It would also give researchers resources with which to attract and train a larger number of graduate students.

The proposed initiative would deliver the needed significant improvements to software quality only if expected research results can be effectively transitioned into practice. There is currently a critical national need for effective technology transition vehicles. This initiative shares that need, but must also share in the costs associated with learning how to meet the need. We echo the commonly heard plea for the formation of university-industry consortia. We believe that such partnerships can facilitate the development of research prototypes, their rapid evaluation, the facilitation of effective improvements, and the expedited use of early production versions of the prototypes. The proposed initiative must provide incentives to encourage universities and industrial organizations to form these sorts of coalitions.

Finally, it should be noted that software is increasingly counted upon in life critical embedded systems. Software errors here can cost human lives and have the potential to cause disasters. The cost of this initiative will seem small indeed on a day when faulty software is determined to have led to a nuclear reactor accident, a commercial aviation disaster, or the accidental initiation of hostile military action. The proposed initiative could significantly reduce the risk that such a day will ever come.

5 Organization of the Workshop

In March 1988, the Program Committee for the Second International Workshop on Software Testing, Verification, and Analysis (TAV2) met in Los Angeles. In addition to deciding the

program for TAV2, the Program Committee also spent a day discussing whether the research community is effectively meeting its challenges. The strong consensus of the group was that the challenge of providing technologies to support effective software testing and analysis is not being met.

The committee's members felt that current research activities in this area are too fragmented, are being conducted on a scale that is generally far too small, are inadequately funded, are insufficiently focused on the larger challenges and responsibilities, and are being conducted by a community that is too small, that exhibits an insufficient sense of community, and that may be in danger of stagnating.

As a first step towards redressing these shortcomings, those in attendance (L.A. Clarke, R.A. DeMillo, S.L. Gerhart, M. Hennell, W.E. Howden, J. Laski, L.J. Osterweil, D. J. Richardson, and L.J. White) agreed to form a group, now known as the Alexander Group, to help coordinate efforts to improve the condition of the software testing and analysis research community.

The existence of the Alexander Group and a brief statement of its goals were announced at TAV2, and a short birds-of-a-feather meeting was held. Over 40 people, representing both researchers and industrial practitioners, attended the meeting. The attendees agreed strongly with the statement of shortcomings expressed by the Alexander Group. They added that efforts to transfer research technologies into industrial practice are currently quite ineffective and that efforts to put such technology transfer mechanisms in place are currently inadequate. It was suggested that it would be important to hold an intensive 2-3 day workshop to help the software testing and analysis community to 1) better articulate its goals, 2) better understand the needs and desires of those who increasingly rely on the community to supply critical technologies, 3) identify promising research directions whose accelerated support would be highly rewarding, and 4) begin planning for initiatives that would strengthen the research community and bring research results into industrial use more rapidly and effectively.

This idea was discussed more widely after the birds-of-a-feather meeting at TAV2 and continued to generate support and enthusiasm. Key researchers from communities such as software testing, software safety, software verification, and software security, expressed interest and support for such a meeting and such initiatives. Industrial software practitioners and managers also showed interest in the idea. Because of this broad based support, the Alexander Group proposed holding a three-day workshop. Discussions with personnel from the Naval Research Laboratory and the Office of Naval Research resulted in sponsorship for the workshop by the Office of Naval Research and the Office of Naval Technology.

The primary goal of the Workshop was to address the issue of how the research community can become significantly more effective in providing technologies to support software testing and analysis. Workshop attendance was limited to 40 people, some of whom were leaders in research and some of whom were knowledgeable in the problems of effective transfer of such technologies to industrial practice. In addition, representatives from key

government research laboratories and funding agencies attended as observers.

Planning for the Workshop was done by an Organizing Committee consisting of Leon Osterweil (Chair), Lori Clarke, Lou Chmura, Richard DeMillo, and Ralph Wachter. The Organizing Committee solicited position papers from the testing and analysis community by means of announcements posted at major software conferences and workshops and by direct mail solicitation, using mailing lists of attendees at past testing and analysis workshops. Approximately 55 position papers written by over 70 authors were received. The Organizing Committee then selected 35 individuals to participate in the workshop and 8 others to attend as observers.

The Workshop consisted of plenary sessions on:

- major accomplishments to date,
- current obstacles and impediments to progress,
- research objectives,
- major initiatives, and
- a workshop wrapup.

During these sessions, all attendees discussed issues as a group. Each session began with a very short formal presentation by a discussion leader, who then led subsequent discussion. Sessions were held in mornings and late afternoons to allow attendees to use the early afternoons for informal small group interaction and consensus formation. Discussion recorders were appointed to provide written summaries of the sessions.

At the conclusion of the Workshop, the Organizing Committee began to draft this White Paper summarizing Workshop consensus and recommendations. An initial draft of the White Paper was circulated among the members of the Organizing Committee. Once this draft was agreed upon, the draft was circulated to all of the workshop attendees, with a cover letter inviting attendees to suggest any changes that they thought needed to be made. Attendees were advised that the Organizing Committee would consider the suggestions and use them as the basis for a revised version of the White Paper. The Organizing Committee indicated that it would circulate this revised version of the White Paper to the wider community of people interested in software testing and analysis. Workshop attendees were allowed a period of 30 days in which to transmit comments to the Organizing Committee. As responses were slow in arriving, the Organizing Committee continued to accept responses for approximately 90 days.

During the response period seven attendees forwarded suggestions for changes to the White Paper. There were no objections to the proposed procedure for developing the second draft of the White Paper. All but one of the respondents were generally supportive of the White Paper and made suggestions for changes to a wide variety of aspects of the White Paper. These changes were, essentially without exception, incorporated into the second

draft of the White Paper, which appears here. One respondent had major reservations about the tone, substance, and recommendations of the first draft. This respondent's suggestions were not incorporated into the second draft, as they were in conflict with the supportive comments of the other respondents. We informally contacted several other participants, all of whom indicated their support for the White Paper verbally. The Organizing Committee took the lack of further replies from other attendees as being tacit approval of at least the general outlines and conclusions of the first draft of the White Paper, and hence tacit approval of the substance of this second draft of the White Paper.

Consistent with the Organizing Committee's promise to promulgate the second draft of the White Paper more widely, we are now publishing this second draft here in order to expose the findings and recommendations to further public review. We hope that this will lead to further attempts to implement Workshop recommendations.

Appendix: List of Participants and Observers

W. Richards Adrion, University of Massachusetts
Bill Brykczynski, Institute for Defense Analyses
John Cherniavsky, Georgetown University
Louis Chmura, Naval Research Laboratory
Lori A. Clarke, University of Massachusetts
Richard A. DeMillo, Purdue University
Laurie Dillon, University of California, Santa Barbara
Stuart I. Feldman, Bellcore, Inc.
John Gannon, National Science Foundation and University of Maryland
Susan L. Gerhart, Microelectronics and Computer Corporation
Donald Good, Computational Logic, Inc.
Richard Hamlet, Portland State University
William Hetzel, Software Quality Engineering
William E. Howden, University of California, San Diego
Richard Kemmerer, University of California, Santa Barbara
Janusz Laski, Oakland University
Nancy Leveson, University of California, Irvine
Richard J. Lipton, Princeton University
David C. Luckham, Stanford University
Brian Marick, Motorola
Rhonda J. Martin, Purdue University
Aditya Mathur, Purdue University
Mark Moriconi, SRI International
Leon Osterweil, University of California, Irvine
Thomas Ostrand, Siemens Research Center
Debra Richardson, University of California, Irvine

James G. Smith, Office of Naval Research
Linwood Sutton, Naval Ocean Systems Center
K.C. Tai, North Carolina State University and National Science Foundation
Richard N. Taylor, University of California, Irvine
Tad Taylor, Computational Logic, Inc.
Jane Van Fossen, Office of Naval Technology
Ralph Wachter, Office of Naval Research
Abe Waksman, Air Force Office of Scientific Research
Robert Westbrook, Naval Weapons Center
Elaine Weyuker, New York University
Lee J. White, Case Western Reserve University
Jack C. Wileden, University of Massachusetts
Steven Zeil, Old Dominion University