General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by NASA STI Compliance and Distribution Services

77-1384

APPROACHES TO SOFTWARE VERIFICATION WITH EMPHASIS ON REAL-TIME APPLICATIONS

J. Barton DeWolf* and Jonathan Wexler
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139

ABSTRACT

Software verification is a critical element in attaining reliable and responsive software. This paper discusses current approaches to software verification in the context of an idealized development life cycle, and surveys current research towards new approaches. Although proof of correctness approaches are included, the focus is on testing technology. Management aspects are covered, as are automated tools to aid the testing process. The real-time emphasis adds a concern for performance evaluation and environmental interaction as well as functional correctness. An underlying theme is the relation between system requirements and the testing, integration, and acceptance process.

1. INTRODUCTION

A recurrent theme in the literature is the high cost and poor quality of computer software. Experience provides far too many examples of software-based systems that fail to operate as the end-user desires.

Concerns about quality software are magnified in the realm of real-time systems; a real-time computer system can be defined as an embedded part of a system which is controlling, measuring, and interacting with some physical device or process. A software-induced failure in a real-time system can produce loss of control, invalid control, or lost data.

Such software problems can be traced to a) the intrinsic complexity of real-time systems, b) poor management of the specification, development and testing processes, c) inadequate requirements analysis and design of systems, d) the inadequate early consideration of testing requirements and e) the intrinsic difficulty associated with testing real-time software. These five sources are different facets of one irreducible fact - these systems are too "large" for complete comprehension by any single person.

Approaches to solving these problems can be broken down into a) strategies for managing the specification, production and testing activities, b) regimens and standards for these production activities and c) techniques for proving/testing software code and system requirements. All of these attempt to make quality assurance a more integral part of the software development process.

This paper is intended to be a survey of new approaches to testing. Its starting point is an overview of current good practice and its shortcomings. This is followed by a discussion and assessment of new approaches.

* Member AIAA

2. THE SOFTWARE LIFE CYCLE AND INTRODUCTION OF ERRORS

2.1 Software Life Cycle

The various phases that software goes through, from the conception of the idea through the operations and use of the software, are termed the software life cycle. This concept is important due to the correlation between the kinds of errors that exist in software, and the various phases of the life cycle. The software life cycle is, thusly, a model of software development and use in two senses. First it is a model in the sense of an ideal, that is, "this is how software should be developed/used;" secondly, it is a model in the sense of an abstraction, that is, a range of activities that have been gathered together and categorized (thereby destroying some of the grey areas between the activities).

Figure 1 depicts the view of the software life cycle that this paper will take. (Other views exist, certain of which incorporate aspects which are beyond the scope of this paper. For instance, specialized electronics may be developed in parallel with the software, leading to a separate integration phase.)

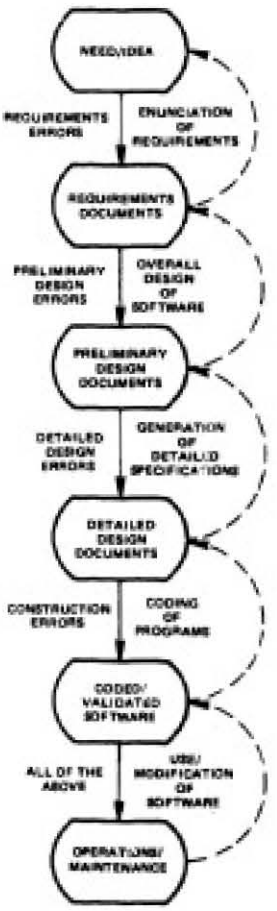


Figure 1. Errors/Introduction in the Software Life Cycle

Copyright © American Institute of Aeronautics and Astronautics, Inc., 1977. All rights reserved.

The first phase starts with a conceptualized need or idea, and consists of the enunciation of requirements. The result of this phase is a requirements document which lays out the functions that should be performed by the software. The process of enunciating is, by definition, a heuristic process. One very difficult aspect of this phase is trying to specify requirements completely and thoroughly, without overly constraining the later phases of the life cycle[1,2].

The next phase is the preliminary design phase, the output from which is a document from which a more detailed design can be derived. Preliminary design is an attempt to derive an overall framework for implementation. At present, this is primarily a heuristic process, although preliminary design is a candidate for automation.

Preliminary design leads to the next phase, the detailed design phase. During this phase, documents are developed which will permit the coding of software. If this phase, and the preceding two phases, have been performed without error, then the detailed design documents will reflect a complete, accurate implementation of the original, conceptual idea with which the life cycle was begun. The detailed design phase is primarily a heuristic process, but also is a candidate for automation.

The next phase is the coding, construction, checkout and integration phase. In this phase, the detailed design documents are used to construct code, checkout the coded software, and put the code together in some sort of finalized package. This phase is partly manual, partly automated. In real-time systems where high reliability is required, the testing aspect of this phase can account for 40-70 percent of total software development cost.

The final phase is the operations and use phase. In this phase the software is used. It is here that software errors have their impact by causing incorrect operation or system failure. It is during the operations and use phase that such errors are found and corrected, and that the software is made to more closely conform to the user's original, or updated, needs.

2.2 Testing Characteristics

As later sections will expound, successful testing requires advanced planning, design of test procedures, monitoring of test results and management of test schedules and required resources. Early consideration of testing, that is, during the requirements and preliminary design phases, is essential to successful testing.

The testing of software, both in actual practice and the theoretical literature, has centered on verifying that the produced software code performs in accordance with the detailed design documents and such testing forms the focus of this paper. However, testing can be performed to check the results of the previous phases; that is, check for inconsistencies in requirements, check that preliminary design does not preclude implementing all of the required functions, etc. Indeed, data gathered on software errors in large systems indicate that design errors constitute the majority of errors that remain undetected at the end of the coding, construction, verification and integration phase.

2.3 Types of Errors

The errors that exist in software can be traced to the individual phases of the software life cycle. This tracing is an important concept in regards to testing, inasmuch as it suggests both a) the range of development activities which are potential error sources and hence, candidates for test and b) the limited amount of assurance which can be had from any single test or testing technique.

Requirements errors are those introduced during the requirements phase and result from any inaccurate expression of the user's needs. These errors can result from the user's inability to understand or express his needs, or the inability of those writing the requirements specification to capture this information.

Preliminary design errors consist of generating an inadequate, erroneous or incomplete preliminary design. Inadequate designs are those that are incapable of performing the required functions using the available resources; that is, designs which will fail to meet performance standards (defined later). Erroneous preliminary designs are those that do not properly implement all the functions, or fail to answer certain questions that should be addressed in a preliminary design. Errors made in the detailed design phase are similar to those made during preliminary design, only they first appear in the detailed design documents.

The label "coding errors" is taken to encompass a broad category of errors, including errors made in generating code, translating code from one language to another and integrating various pieces of software. In short, an error during the coding/construction/checkout/integration phase is any error which causes the coded software to not behave exactly as the detailed design documents indicate.

Also, during one of these phases, the final "user's manual" will be produced. An error in the user's manual - that is, a misleading instruction to the user - can cause serious operational difficulties.

2.4 A Definition of Testing

Hetzel, in Program Test Methods[3], defines testing as "any activity that increases our assurance or belief that a program performs 'as it should.'" The goal of testing is to make sure that the produced software code performs, during the operations phase, as desired in the conceptual phase, or as specified in a later phase. In real life, testing focuses primarily on verifying a) lack of coding/construction errors, and b) lack of design and specification errors. The phrases of "making sure" or "verifying" bring up the issue of levels of confidence. While one would

naturally want to obtain a 100% level of confidence that there were no errors in a given piece of software, the cost of this would be prohibitive. This forms another characteristic of testing, that it is finite in nature, and that the incremental cost of further testing must be weighed against the incremental benefits.

3. CURRENT APPROACHES TO TESTING

3.1 Good Current Practice

This section is based on a consensus in the literature concerning good practice in the realm of software testing, as well as discussions with experienced test engineers.

Figure 2 shows the testing process during the software life cycle and depicts the milestones achieved during the testing process. The software life cycle should appear as the backdrop in this figure, but has been omitted for reasons of editorial clarity.

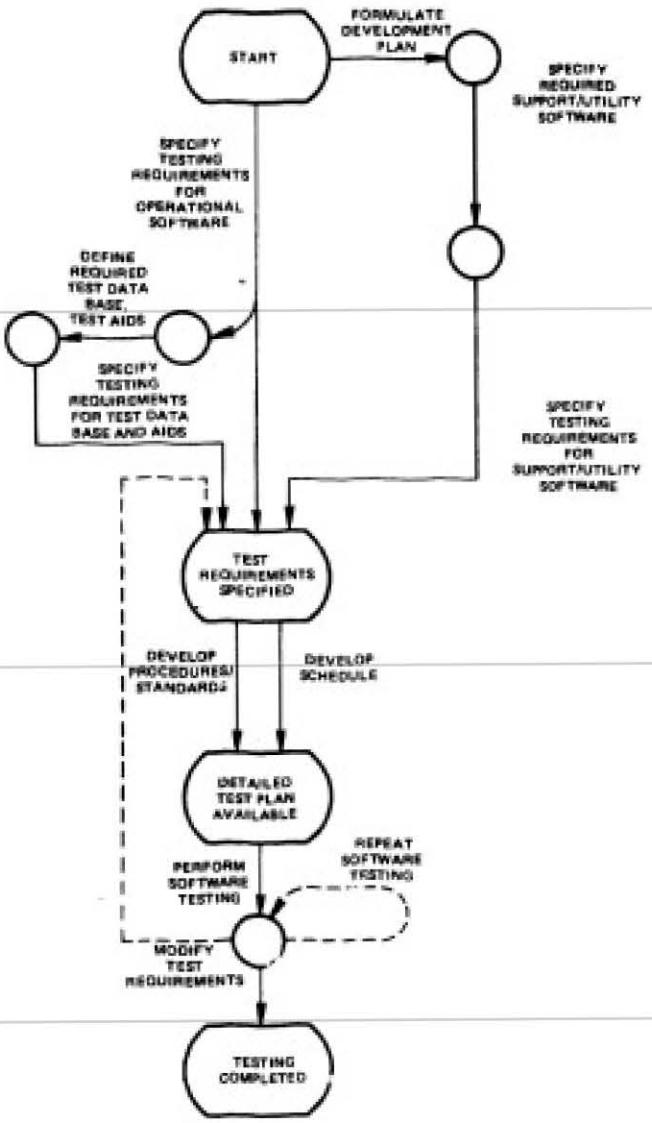


Figure 2. The Testing Process

The test planning process must start during the requirements phase (at the time when requirements are stated in testable form), but cannot be completed until the preliminary design phase, when details of a test plan can be committed to paper. These details are known as test procedures, and lay out the sequence of manual/automated actions which comprise a single test.

At this point, a few definitions would be helpful. "Operational software" is the desired end-product of the development effort; an example of this would be navigation software found aboard an aircraft. "Support/utility software" is generally taken to include a broad range of general purpose software, such as compilers, assemblers, loaders, file utilities and batch and timesharing operating systems. A "test data base" is a loosely defined term; it means an aggregate of data (simulated or real) which can be used to test software. For instance, to test software that processes radar signals, it would be helpful to have digitized radar signals to be used as a test case. This data might, in turn, be simulated data; that is, synthesized by another computer program. This introduces a fourth category of software: test aids. These test aids are designed to simplify the testing procedures, and include all of the following: a) software to generate test data; b) software to analyze the results of testing; c) simulators to produce benchmark results; d) test drivers/"stubs" to provide "environment" for the software under test; e) configuration control tools; and f) testing/debugging tools.

Testing/debugging tools act as a supplement to manual methods such as desk checking. Interactive debugging tools are designed to give the programmer greater visibility into the actions of his software, by permitting breakpoints to be set and data to be altered and displayed. Compilers can perform analyses of code, and indicate testing levels, argument mismatches, "unreachable" statements, etc. Analyses of a dynamic nature can be performed, such as accumulating execution counts for various blocks of code. Current research relating to certain categories of these tools is presented in section 4.

Another feature of testing is the development and enforcement of procedures, standards, and schedules. Here the relationship of testing to the life cycle is apparent. The development of these controls is essential to coordination and communication between several groups of people. Enforcement is essential to attainment of the full effect of these procedures.

As Figure 2 shows, software testing is an iterative process. One source of this is that a piece of software may need retesting, owing to a change in that piece, or a related piece, of software. This retesting is one of the central issues in testing, as it may be infeasible to return to the beginning of test plan each time any relevant piece of software is modified. Another source of iteration can be the modification of test requirements/preliminary design/detailed design; these modifications may or may not be a response to the results of tests performed through that time.

Other issues exist. One of these is the issue of "independent testing;" here, separate sets of individuals perform the tasks of software development and software testing, with a well-defined interface between the two. This has advantages in terms of the increased amount of attention given to testing, and the freedom from bias that the testing group enjoys. Disadvantages may exist in terms of cost and scheduling.

There is also the issue of testing strategies. The "top-down" school of testing claims that software should be testing by starting with the higher-level, presumably control-oriented software, and then integrating, piece-by-piece, the lower level presumably computations-oriented pieces. Advocates of a "bottom-up" approach believe that the lower level pieces should be tested first. Both schools believe that testing should be incremental - that is, software should be added, to the aggregate of previously tested software, in tractable pieces. The purpose of this is to help isolate problems. However, neither school has much counsel on what size or shape the increments should be.

Another important notion is that of design for testability. If consideration of a test plan is delayed until the detailed design phase, it may be too late to develop a straightforward, effective test plan - testability may wind up "designed out." The need here is for explicit, well-structured requirements specification. This allows early attention to the demands of testing.

The design or selection of programming languages, while an important design consideration, also has an impact on testing. "GO TO" - free coding techniques, restricting programmers to a subset of a programming language, redundant specification, etc., all claim advantages in reducing the frequency with which software errors are made, and easing the correction of such errors.

Another important aspect of testing is related to demonstration and documentation of results. Recording facilities, such as an ability to print out test results, are a must. Repeatability of tests is crucial to being able to demonstrate the proper functioning of software. The documentation produced should include the following: first, the overall test plan, which should state the goals, standards, and philosophy of the test program, in addition to a list of test procedures. These test procedures should detail the acts to be taken to perform a given test; for each such test, the actual results should be documented.

3.2 Problem Areas

Certain issues are not answered, or not very well answered, by current state-of-the-art approaches. Several problems arise as a result of a lack of criteria for test case selection and evaluation of test effectiveness. One such question is to how much to test, with the quantity of testing being defined both in terms of how many different test procedures and how many test cases to be run with any given test procedure. Also, there is no well-developed way of generating test data with critical data values.

Acceptance testing, particularly third-party testing, remains a serious problem. Most testing approaches assume a detailed knowledge of program structure which the acceptance tester is unlikely to possess. Acceptance testing resembles "black box" testing which is notoriously unreliable as a strategy for program checkout. A good requirements document is essential for the acceptance tester, which raises yet another problem area. Current practice does not do much in the area of testing requirements, testing preliminary design or testing detailed design. This undoubtedly accounts for the preponderance of design errors over coding errors in delivered software. Measuring/estimating software quality is also problematic.

Currently, an effective test plan requires a voluminous, and costly, amount of testing. The reduction of testing requirements is a problem of current interest. Finding "better" ways to test, e.g., avoiding costly simulations, is also a problem. Another costly aspect of testing is the need to retest software that has been modified; isolation of the impact of such changes is the problem.

4. CURRENT RESEARCH AND NEW APPROACHES

This section will review briefly some of the research currently underway to improve the process of software test and verification. It is divided into the following five areas: (1) static code analysis, (2) dynamic code analysis, (3) test case generation, (4) program correctness proofs, and (5) statistical quality estimation. Highlights of current work in each of these areas are presented together with literature references (see also the reviews by Stucki, et. al.[4], and Ramamoorthy and Ho[5], and the survey of tools by Reifer and Trattner[6]). A final section presents some opinions with respect to how well current research is addressing the problems identified in the previous discussion.

4.1 Static Code Analysis

Code analyzers have been subdivided into the two categories: static analyzers and dynamic analyzers. The distinction is that dynamic (execution) analyzers require execution of the code, whereas static analyzers do not. Static analysis can be regarded as an augmentation of normal compiler or assembler diagnostic functions, but without the generation of object code. In this category are included standards checkers, data usage analyzers, cross reference table generators, path analyzers, and a variety of other tools. Static analysis is usually the first step in automatic test case generation, considered separately in 4.3 below.

Table 1 lists a number of static analyzer examples, the input language on which they operate, the functions performed, literature references, and the corporation or university at which the tool was developed. Some of these tools, like DAVE or FACES, provide functions in addition to

Table 1. STATIC ANALYZER EXAMPLES

TOOL	INPUT	FUNCTIONS	REF	SOURCE
DAVE	FORTRAN	Structural analysis; data flow and usage checks; error-prone constructs checking; documentation.	7	U. of Colo.
FACTS	FORTRAN	Software quality checks; error-prone constructs, interfaces, program structure, standards; user requests and documentation aids.	5,8	U. of Calif., Berkeley
PACE	FORTRAN	Some static analysis; mainly dynamic analysis.	9,10	TRW
PFORT Verifier	FORTRAN	Checks for adherence to a portable subset of ANSI FORTRAN	11	Bell Labs
RXVP	FORTRAN	Software quality checks and structural analysis; static statement-type usage statistics; also dynamic analysis.	12,13	GRC
Sneak Circuit Analyzer	Assembly Language	Analyzes for error-prone constructs using an approach developed for electrical circuit analysis.	14	Boeing Aerospace

verification aids, such as information retrieval or documentation. Some, like PACE and RXVP, have both static and dynamic analysis capabilities. Some are experimental, others are prototypes or program products. Most of these tools have been developed for ANSI FORTRAN in the interests of portabilitiy; some, in fact, check for deviations from the standard or the use of other machine-dependent language features.

Table 1 provides a sampling of some current analyzers; many others exist or are under construction. For example, a static analyzer is being constructed at the Draper Laboratory to examine assembly language real-time flight code for the proper insertion of roll-back points and the checking of pointer variables. The static analysis of multi-process real-time code is a fruitful area for future investigation.

4.2 Dynamic Code Analysis

Dynamic analyzers provide information on the characteristics of a program under execution. They perform such functions as detecting the traversal of various program path segments, monitoring the frequency of segment execution, determining maximum and minimum values assigned to variables, and so forth. One of the most important functions is evaluation of the effectiveness of a set of test cases by determining how much of the program has been exercised.

Table 2 lists a few of the dynamic analyzers which have been developed. While a few of these analyzers (notably, PPE) operate on object code and hence are language independent, most function by instrumenting the source code with data-collection probes which build an execution statistics data base for post-run analysis (see Figure 3). The source language for most of these analyzers is FORTRAN, though a few have been

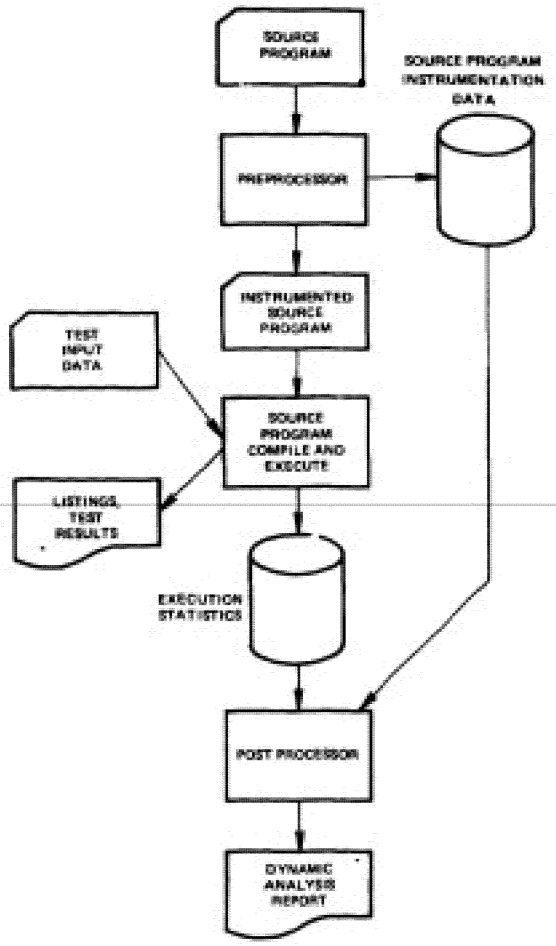


Figure 3. Dynamic Analyzer Operations Flow

Table 2. DYNAMIC ANALYZER EXAMPLES

TOOL	INPUT	FUNCTIONS	REF	SOURCE
PACE (FLOW, TDEM)	FORTRAN	Instrumentation of source code, run-time statistics; evaluation of testing effectiveness.	9,10	TRW
PET	FORTRAN	Instrumentation of source code, run-time statistics; code utilization; location of dead code, impossible branches; listing of values assigned to variables and indices.	4,15	MDAC
PLUM	PL/1	A compiler-based analyzer to generate statistics on program size, characteristics, and error history.	16	U. of Maryland
PPE	Object Code	Performance monitoring; statistics on timing, program state changes, efficiency.	4	Boole & Babbage
RXVP	FORTRAN	Instrumentation of source code, run-time statistics; code utilization; listing of values assigned to variables; evaluation of testing effectiveness.	12,13	GRC
SMT	Object Code	Instrumentation of object code, run-time statistics and analysis.	17	U. of Calif., Berkeley
Others	COBOL, JOVIAL, PL/1	Similar to above	4,6	

developed for other languages including PL/1, COBOL, and JOVIAL. The PLUM analyzer listed in Table 2 is different in that it is compiler-based and hence produces statistics on all programs using the facility.

One of the newer functions being tackled by dynamic analyzers is the checking during execution of programmer generated assertions[15]. This has promise as a method for relating program specifications in the form of assertions to test data gathering and analysis. The assertions can be either local to a specific point or region of code, or global. The presence of the assertions in the code also helps to document the intent of the designer and should be useful during program maintenance.

4.3 Test Case Generation

The static and dynamic analyzers described above provide information about program structure and behavior. Several of them have been extended to provide assistance in test case generation or selection. The problem of automatically generating test data is closely tied to the problem of determining reliable and valid testing criteria. Neither problem has a satisfactory solution as yet, though some progress has been made.

Various criteria have been proposed for test selection (see Table 3). The first criterion — programmer judgment — is most frequently selected by default. The last criterion — test all legal paths -- usually implies an impossibly large, often infinite, collection of tests. The third criterion — execute all branches — implies the second except for unreachable code and is the one which has received most attention from the point of view of automatic test case generation.

Table 3. TESTING CRITERIA

1. Programmer judgement.
2. Execution of all program statements.
3. Execution of all program branches.
4. Dividing program paths into equivalence classes, and execution of at least one path from each class.
5. Execution of randomly-selected test data.
6. Execution of all legal program paths.

Table 4 lists and describes some examples of test case generators and serves to illustrate the various implementation philosophies. Most of these tools start out with a static analysis of the program under test to identify program paths and branching structure. They then proceed to select test paths according to some criterion (sometimes with the programmer's help), and then proceed either backwards or forwards along the path, evaluating the branching predicates symbolically and forming inequality constraints associated with path execution. Sometimes a selected path can be eliminated as logically impossible. In other cases the inequality constraints can be solved to produce a range of input values associated with the path so that test cases can be selected. In still other cases, the inequality constraints are too complicated to be solved — they are often nonlinear — and the programmer must be called in for assistance. The techniques based on symbolic execution (DISSECT, EFFIGY, and SELECT, for example) have additional capabilities: they can be used to symbolically evaluate intermediate or final results along a path based on a range of input values. These results can be manually inspected for correctness, or automatically compared with programmer-supplied assertions. Although promising, the techniques are still rather limited in terms of the program size and complexity with which they can deal.

A more fundamental problem is the reliability and validity of the testing criteria (Table 3) themselves. How trustworthy are these criteria in detecting program errors and verifying correct behavior? Goodenough and Gerhart[26] have provided the beginnings of an answer to this question by first introducing some helpful semantics (see Table 5). Reference 26 should be consulted for a more precise statement of their result. Their definitions of reliability and validity and the related theorem have inspired a deeper understanding

Table 4. AUTOMATIC TEST CASE GENERATOR EXAMPLES

AUTHOR/TOOL	INPUT	DESCRIPTION	REF	SOURCE
Bayer, et. al./ SELECT	LISP subset	Uses symbolic execution to generate input need to execute a selected path; accepts assertions.	18	SRI
Clarke (builds on DAVE	FORTRAN	Defines possible paths; symbolic execution is used to create inequality constraints on input variables which are then solved.	19	U. of Mass. Amherst
Hoffman, et. al./ ADTG (builds on PACE/TDEM)	FORTRAN	Constructs paths which exercise each branch at least once; based on a directed graph, connectivity matrix.	20	TRW
Howden, Stucki, et. al.	FORTRAN	Generates classes of paths and selects one case from each class; evaluates predicates symbolically then uses inequality solution techniques and a backtrack search.	4	MDAC
Howden, Stucki, et. al./DISSECT	FORTRAN	User selects a path; symbolic execution is used to evaluate the outputs; includes some assertion checking capabilities	4	MDAC
King/EFFIGY	PL/1 dialect	Interactive symbolic executer; user-selected input path is symbolically executed; checks assertions.	21	IBM
Miller, et. al. (builds on RXVP)	FORTRAN	Determines execution paths which exercise each predicate; constructs test cases using a backtracking technique.	22	GRC
Ramamoorthy, et. al. (builds on FACES)	FORTRAN	Generates test data using symbolic execution.	23	U. of Calif. Berkeley

Table 5. THE SEMANTICS OF TEST DATA SELECTION (REF. 26)

(1) The outcome of a set of test cases is successful if the program produces correct output for each of the cases in the test case set.

(2) A test criterion is reliable when it produces a set of test cases such that the outcome is consistently successful or unsuccessful for a given program.

(3) A test criterion is valid for a given class of errors when it produces at least one set of test cases capable of revealing the error.

Theorem. The successful execution of a complete set of test cases selected according to a criterion that is both reliable and valid for a given class of errors is sufficient to prove the absence of that class of errors from a program.

of the testing process and a search for better testing criteria[27,28]. More work along these lines is badly needed.

4.4 Program Correctness Proofs

The literature on proving programs correct is immense and the subject complicated. The reader is referred to the reviews of Elspas, et. al[29] and London[30]. Following Elspas, the approaches can be broadly categorized as informal versus formal. In the informal approaches, the programmer annotates his code at various points with assertions regarding the expected or intended state of program variables. Alternatively, the assertions can be formulated first, following which the code is developed in a top-down manner. Informal logic is used manually to verify that the assertions do indeed hold for a given input assertion and set of program statements. The formal approaches differ in that the process of verifying assertions is transformed into the proof of a theorem in some system of mathematical logic, usually the first-order predicate calculus. The goal of researchers in this area has been the development of automatic theorem-provers based on formal logic to assist the programmer in carrying out proofs of correctness. This goal is still largely unrealized for realistic programs either because the formalism for expressing assertions is difficult to use, and/or the proof technique requires too much computer time and memory. Examples of contemporary interactive program proving systems together with an assessment of the state-of-the-art can be found in the papers by Good, London, and Bledsoe[31], and Hantler and King[32].

Most proof-of-correctness approaches have been concerned with the non-real-time aspects of a program under development: namely, that it terminates and produces a correct result given any allowable input. New work in this field is beginning to address timing and synchronization properties of a program system. It is not yet clear whether proof techniques can be developed for real-time systems which substantially alleviate the testing burden associated with assuring correct real-time behavior. This also appears to be an area of high potential for future research.

4.5 Statistical Quality Estimation

Statistical quality estimation refers to a variety of techniques which attempt to infer characteristics of the software product (such as reliability) from statistical information gathered during the development process. Two techniques of particular interest are error modeling and error insertion.

Error modeling can be used to provide an estimate of the number of errors remaining in a unit of software on the basis of an error-removal history maintained during the development cycle. Two good recent papers are those of Moranda[33] and Shooman[34], who provide references to other literature. The technique is based on the plausible hypothesis that the rate of error detection and removal is proportional to (or is some other function of) the number of errors remaining; thus, a measure of the former together with a model of the error detection process can provide an estimate of the latter. Error detection rates are usually measured as the number of errors detected per unit of computer time used for testing, to compensate for variations in the level of testing activity. Existing data from a wide variety of software development projects appears to support the basic assumptions of this approach, though a variety of error models are currently advocated.

The error insertion technique is based on a similar hypothesis, but has a strong advantage in that the parameters of the error detection process are directly measured. A good description of this technique is in Gilb's recent book[35]. In this approach, the software under test is seeded with a known number of "representative" errors, and error detection data are kept. At any point in time, the fraction of real errors found is assumed to be equal to the fraction of seeded errors found, so that an estimate can be made of the number of real errors remaining. A difficulty with this procedure is assuring that the artificial errors are indeed representative of the real errors to be found in the program, and a careful selection of error types is required. More experience is needed before the practicality of these statistical techniques can be judged.

4.6 Assessment of Current Research

Table 6 summarizes our view of how well current research has been addressing those topics identified in section 3.2 as problem areas with current testing practices:

- the lack of test criteria and effectiveness measures,
- the difficulty of acceptance testing,
- the persistence of requirements and design errors,
- quality assessment,
- the isolation of change impact,
- the high cost of testing and simulation.

Table 6. ASSESSMENT OF CURRENT APPROACH

Research Area \ Problem Areas	Test Criteria and Effectiveness Measures	Acceptance Testing	Requirements and Design Errors	Quality Assessment	Isolating Change Impact	Cost of Testing and Simulation
Current						
1. Static Code Analysis	S	S			S	S
2. Dynamic Code Analysis	S	S			S	S
3. Test Case Generation -Testing Theory	S P	S			S S	S P
4. Program Correctness Proofs					S	S
5. Statistical Quality Estimation				P		S
Needed						
6. Theory of Requirements	S	P	P		S	S
7. Hierarchical Testing Approaches		S	S		P	S
8. Real-time Assertions/ Proofs	S	S	S		S	P

P - primary impact; S - secondary impact

A P in the table indicates that the research area might be expected to have a primary impact on the problem area, whereas an S indicates a secondary impact.

As the table indicates, it is our view that current research will have a primary impact in only a few of these problem areas, though secondary impacts are more widespread. The problem of rooting out requirements and design errors is not addressed by any of the research areas mentioned, for example, though some research is underway in the area of software requirements and design theory. The reason for this somewhat pessimistic view is that the problems are very difficult and not readily amenable to theoretical or systematic technical approaches, at least at our present level of understanding.

Can research be undertaken which might more adequately address some of these problem issues? Three areas are proposed in Table 6. These are:

a) Theory of requirements - attempt to build a more direct relationship between software requirements and software testing; attempt to formalize the requirements and check for completeness and consistency.
b) Hierarchical testing approaches - improve the ability to isolate change impact through a study of the testing of hierarchically-ordered software systems.
c) Real-time assertions/proofs - reduce the need for testing and Monte Carlo simulations by analytically establishing certain real-time properties: interrupt structures, absence of deadlock, timing properties, etc.

These research topics would no doubt build on elements of current research to increase our understanding of the verification process.

5. CONCLUSIONS AND RECOMMENDATIONS

This paper has described aspects of current practice and current research related to software verification, particularly the verification of real-time computer systems. The software life cycle has been described, and various categories of software errors identified. A good-current-practice view of the testing process reveals that many problems remain in detecting and correcting software errors. A review of on-going research suggests that several of these important problems are not being adequately addressed by current research efforts. Some additional categories of needed research have been suggested to supplement the existing activity.

Research in the area of software testing should aim to provide a better theoretical/empirical base for the testing process. This includes efforts to establish a theory of software requirements, and to forge a stronger link between the requirements process and testing. Better test criteria and effectiveness measures are also needed. Quality measures and assessment techniques are being proposed for software, but more applications experience is required before their value can be judged.

Finally, let it be emphasized that the best way to rid a software system of errors is still to prevent their introduction in the first place. A better job in the requirements and design phases of the life cycle clearly pays off when it comes to testing. Although not emphasized here, much can be done during design to reduce the incidence of design errors and to facilitate the testing process. Better design procedures and guidelines are strongly recommended to help in achieving the goal of improved software reliability.

References

1. DeWorkin, D. and DeWolf, J.B., "Critical Issues in Data Processing Requirements Specification," R-1080, May 1977, The Charles Stark Draper Laboratory, Inc., Cambridge, Mass.

2. DeWolf, J.P., "Requirements Specification and Preliminary Design for Real-Time Systems," Proc IEEE International Software and Applications Conference, Chicago, 1977, IEEE Computer Society, Silver Spring, Maryland.

3. Hetzel, W.C., ed., Program Test Methods, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

4. Stucki, L.G., et. al., "A Methodology for Producing Reliable Software," Vols. 1 and 2, N76-29946, May 1976, McDonnell-Douglas Astronautics Co., Huntington Beach, Calif.

5. Ramamoorthy, C.V., and Ho, S.F., "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, Mar. 1975, pp. 46-58.

6. Reifer, D.J., and Trattner, S., "A Glossary of Software Tools and Techniques," Computer, Vol. 10, No. 7, July 1977, pp. 52-60.

7. Osterweil, L.J., and Fosdick, L.D., "DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs," Software - Practice and Experience, Vol. 6, No. 4, Oct.-Dec. 1976, pp. 473-486.

8. Ramamoorthy, C.V., and Ho, S.F., "FORTRAN Automatic Code Evaluation System (FACES)," Part 1, AD/A-004168, July 1974, Electronics Research Laboratory, University of California, Berkeley, Calif.

9. Brown, J.R., et. al., "Automated Software Quality Assurance," in Hetzel, W.C., ed., Program Test Methods, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973, pp. 181-203.

10. Brown, J.R., and Hoffman, R.H., "Evaluating the Effectiveness of Software Verification - Practical Experience with an Automated Tool," Proc. AFIPS 1972 Joint Computer Conference, Vol. 41, Part 1, AFIPS Press, Montvale, N.J., pp. 181-190.

11. Ryder, B.G., "The PFORT Verifier," Software - Practice and Experience, Vol. 4, No. 4, Oct.-Dec. 1974, pp. 359-377.

12. Miller, E.F., Jr., "RXVP: An Automated Verification System for FORTRAN," Proc. of Computer Science and Statistics, Eighth Annual Symposium on the Interface, Health Sciences Computing Facility, University of California, Los Angeles, 1975, p. 328 (abstract only).

13. Miller, E.F., Jr., et. al., "Structurally Based Automatic Program Testing," EASCON '74 Record, Institute of Electrical and Electronics Engineers, New York, 1974, pp. 134-139.

14. Ranking, J.P., Engels, G.J., and Godoy, S.G., "Software Sneak Circuit Analysis," AD/A-024718, Apr. 1976, Boeing Aerospace Co., Houston, Tx.

15. Stucki, L.G., and Foshee, G.L., "New Assertion Concepts for Self-Metric Software Validation," Proc. 1975 International Conference on Reliable Software, IEEE Computer Society, Long Beach, Calif., 1975, pp. 59-65.

16. Zelkowitz, M.V., "Automatic Program Analysis and Evaluation," Proc. 2nd International Conference on Software Engineering, IEEE Computer Society, Long Beach, Calif., 1976, pp. 158-163.

17. Ferrari, D., and Liu, N., "A General-Purpose Software Measurement Tool," Software - Practice and Experience, Vol. 5, No. 2, Apr-Jun 1975, pp. 181-192.

18. Bayer, R.S., Elspas, B., and Levitt, K.N., "SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution," Proc. 1975 International Conference on Reliable Software, IEEE Computer Society, Long Beach, Calif., 1975, pp. 234-245.

19. Clarke, L.A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, pp. 215-222.

20. Hoffman, R.H., "NASA/Johnson Space Center Approach to Automated Test Data Generation," Proc. of Computer Science and Statistics, Eighth Annual Symposium on the Interface, Health Sciences Computing Facility, University of California, Los Angeles, 1975, pp. 336-341.

21. King, J.C., "A New Approach to Program Testing," in Hackl, C.E., ed., Programming Methodology, Springer-Verlag, New York, 1975, pp. 278-290.

22. Miller, E.F., Jr., and Melton, R.A., "Automated Generation of Testcase Datasets," Proc. 1975 International Conference on Reliable Software, IEEE Computer Society, Long Beach, Calif., 1975, pp. 51-58.

23. Ramamoorthy, C.V., Ho, S.F., and Chen, W.T., "On the Automated Generation of Program Test Data," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 293-300.

24. Huang, J.C., "An Approach to Program Testing," Computing Surveys, Vol. 7, No. 3, Sept. 1975, pp. 113-128.

25. Henderson, P., and Quarendon, P., "Finite State Testing of Structured Programs," in Robinet, B., ed., Programming Symposium, Springer-Verlag, New York, 1974, pp. 72-80.

26. Goodenough, J.B., and Gerhart, S.L., "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 156-173; correction in Vol- SE-1, No. 4, Dec. 1975, p. 425.

27. Howden, W.D., "Reliability of the Path Analysis Testing Strategy," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, pp. 208-215.

28. Miller, E.F., Jr., "Program Testing: Art Meets Theory," Computer, Vol. 10, No. 7, July 1977, pp. 42-51.

29. Elspas, B., et. al., "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, Vol. 4, No. 2, June 1972, pp. 97-147.

30. London, R.L., "A View of Program Verification," Proc. 1975 International Conference on Reliable Software, IEEE Computer Society, Long Beach, Calif., pp. 534-545.

31. Good, D.I., London, R.L., and Bledsoe, W.W., "An Interactive Program Verification System," Proc. 1975 International Conference on Reliable Software, IEEE Computer Society, Long Beach, Calif., 1975, pp. 482-492.

32. Hantler, S.L., and King, J.C., "An Introduction to Proving the Correctness of Programs," Computing Surveys, Vol. 8, No. 3, Sept. 1976, pp. 331-353.

33. Moranda, P.B., "Prediction of Software Reliability During Debugging," Proc. 1975 Annual Reliability and Maintainability Symposium, Institution of Electrical and Electronics Engineers, New York, 1975, pp. 327-332.

34. Shooman, M.L., "Software Reliability: Measurement and Models," Proc. 1975 Annual Reliability and Maintainability Symposium, Institution of Electrical and Electronics Engineers, New York, 1975, pp. 485-491.

35. Gilb, T., Software Metrics, Winthrop Publishers, Inc., Cambridge, Mass., 1977.