THE SOFTWARE DEVELOPMENT SYSTEM: STATUS AND EVOLUTION

Carl G. Davis and Charles R. Vick

Ballistic Missile Defense Advanced Technology Center
Huntsville, Alabama

Summary

The Software Development System (SDS), a research program designed for the development of software for Ballistic Missile Defense software, is evolving toward increased capability in the areas of testing, quality assurance, and system-level definition. This paper discusses the evolution of SDS and plans for future development.

Introduction

The problems involved in detecting and intercepting a threatening Ballistic Missile, or more specifically numerous, simultaneous reentry vehicles, rank among the most challenging technological endeavors of modern times. System complexity is derived from the synergistic effects of ultrarapid response, problem/solution size and complexity, and a nondeterministic problem space. An integrated response from the most sophisticated of modern technology in sensors, kill mechanisms, and computer science/engineering is basic to the realization of such systems. Many such technological responses in data processing have been developed over the past decade by the Ballistic Missile Defense Advanced Technology Center (BMDATC). Among these has been an extensive and advanced effort in software engineering. The software problem is typically (1) extensive, requiring development of up to 1 million lines of real-time code and several times this number in support software instructions; (2) ultraresponsive with timelines in the milliseconds; (3) complex mathematical and logical algorithms which detect and determine the nature of objects in a hostile problem space and direct kill mechanisms toward the destruction of threatening objects in either the atmosphere or outer space; (4) evasive of conventional testing since the software can never be executed in the actual problem space (nuclear effects and massive attacks) until it is actually called upon in an engagement. These, among other issues, are the motivating force behind the collection of software engineering technology derived by BMDATC over the past several years known as the Software Development System (SDS). An overview of SDS, along with problem characteristics and the developmental environment, has been described.[1,2,3,4] Much interest has been demonstrated throughout the entire software community since the initial release of SDS in 1977, and components are being used in a variety of non-BMD applications.[5,6] Recent endeavors in evolving SDS capabilities include new technologies in software testing and quality assurance, extensions to deal with the system requirements interface, and interpretation of the system in terms of distributed data processing. This paper will concentrate upon a discussion of testing and quality assurance extensions to SDS. Extensions needed for systems requirements interfacing and distributed data processing are described in detail in Reference 7.

SDS Description

SDS consists of a sequence of activities designed to promote an efficient unfolding of a system with an emphasis upon early definition of the computational requirements. The research framework which provided the focus of SDS research activities is shown in Figure 1. The previously described[1] characteristics and goals of SDS are summarized in the following.

- SDS was designed for maximum utilization of the computer as a development tool. This is obtained through machine analysis of system descriptions both at the requirements and design level, and through computer-aided approaches to testing, verification and validation.
- SDS strives for early definition of data processing as an integral part of an overall system definition through an emphasis upon decomposition and formalism.
- SDS allows early definition of structure but seeks to avoid premature design decisions through expressive techniques which allow expression and evaluation of solution alternatives and through providing traceability to design decisions when they are made.

The remainder of this paper will discuss the specifics of new technologies applied to the testing, verification, and validation areas of SDS. In complying with the goals of SDS to have demonstrated quality at each level of definition, the following techniques are computer-aided approaches for assisting in testing, verifying, and validating software.

Adaptive Testing

Increased computer assistance in the testing process has been a goal of the SDS and comes about

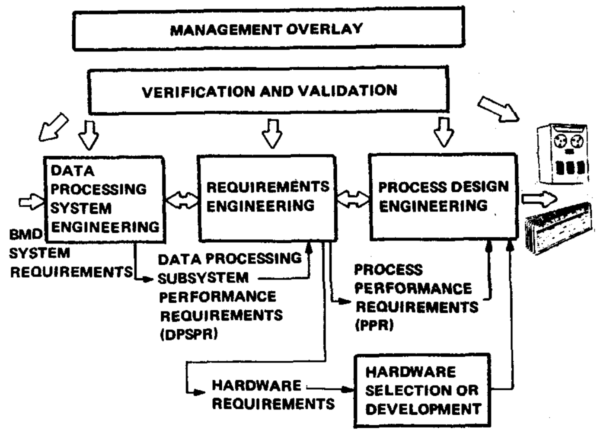

MANAGEMENT OVERLAY
VERIFICATION AND VALIDATION
DATA PROCESSING SYSTEM ENGINEERING
REQUIREMENTS ENGINEERING
PROCESS DESIGN ENGINEERING
BMD SYSTEM REQUIREMENTS
DATA PROCESSING SUBSYSTEM PERFORMANCE REQUIREMENTS (DPSPR)
PROCESS PERFORMANCE REQUIREMENTS (PPR)
HARDWARE REQUIREMENTS
HARDWARE SELECTION OR DEVELOPMENT

Figure 1. SDS Research Framework

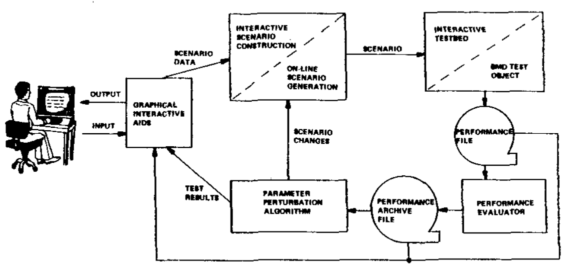
as a natural consequence of the desire to reduce testing costs. Due to the size and complexity of the systems under development, most of the existing techniques that rely on forcing a program to test previously untried paths or segments[8,9] fail in the sense that all paths cannot be exercised. Testing theory is in its infancy,[10] and hence we must look for a practical approach which will derive the maximum information about program operation given a sequence of tests. This adaptive testing approach seeks to generate test case criteria based upon performance and automated techniques for test case generation, data collection and reduction, employs test case perturbation.

In the performance-critical world of weapons systems, the use of performance parameters to evaluate system effectiveness arises from the desire to partition the input space into regions of acceptable and unacceptable performance. If this can be achieved with a reasonable degree of accuracy, then much information could be inferred from such a definition. Questions about performance against expanded or new threats could much more readily be answered. Information about the robustness of a system and the achievement of design goals could also be obtained more readily. This approach could also produce the information which in effect allows an analysis for sensitivity to stress, which has been almost impossible to obtain with large systems.

Computer-aided test case generaion can in itself significantly reduce testing costs. Back-of-the-envelope studies indicate at least 50% savings in testing costs through automation, which translates into a 25% reduction in system development costs. This comes about through efficiencies in test case generation and data reduction and analyses tasks. To be effective, however, this must be an approach that will cause testing to proceed toward a predefined limit (boundary) in an efficient manner. This necessitates information about the system being embedded in an algorithm designed to search a complex performance surface. The combination of an intelligent perturber of test cases and their automated generation also has potential application as a system design aid. If the system parameters rather than the threat are varied, the potential exists to allow the system to be "tuned" against given input conditions. System parameters contributing to poor performance then become readily identifiable and candidates for redesign.

There are several components and activities which must logically exist in an adaptive tester (Figure 2). The test cycle would begin by inputting, with the aid of interactive graphics, a normal threat case well within the anticipated performance boundary of the system. Using a flexible interactive data base builder,[11] the required formats to allow computer-aided test case generation are built. Rapid modification is achieved through defining the test case as a hierarchy of files.

OBJECTIVE: ACCURATELY DETERMINE SOFTWARE PERFORMANCE BOUNDS AND SIGNIFICANTLY REDUCE COST AND TIME FOR TESTING THROUGH AUTOMATED AND ADAPTIVE ANALYSIS



RATIONALE:

- THE MAGNITUDE OF EVEN AN OPTIMAL SET OF TEST CASES RENDERS "EXHAUSTIVE TESTING" UNACHIEVABLE
- NO ORGANIZED ATTEMPTS HAVE BEEN MADE TO DETERMINE, EXPRESS IN CLOSED FORM AND UNDERSTAND THE CAUSES OF SOFTWARE PERFORMANCE LIMITS

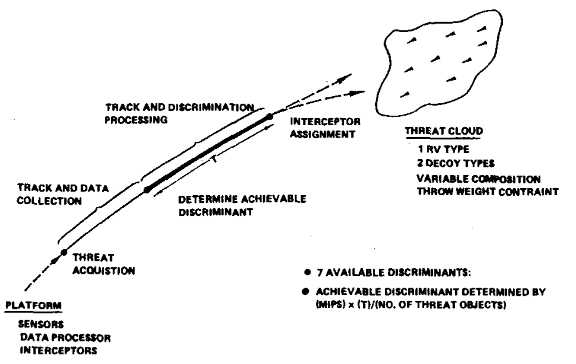
Figure 2. Adaptive Testing

The interactive testbed consists of the process under test and the environment to which it must interact. A properly instrumented process is executed and performance data are recorded for subsequent analysis.

The third activity, performance evaluation, strives to develop an indication of the stress of the system based upon the performance data. Statistical techniques such as factor analysis are used to determine parameters upon which stress parameters are based.

The Parameter Perturbation Algorithm (PPA) provides an intelligent perturbation of the input parameters based upon stress parameters. The PPA is designed to take advantage of previous tests and defined heuristics to decide upon the parameters to perturb. The key to successful search by the PPA is the ability to span the input space with a complete set of heuristics. The loop is now closed through the generation of a new test case, and testing is then repeated in an automated and closed-loop manner until a point on the boundary is reached.

The feasibility of this adaptive testing approach has been demonstrated on a system level test object (Figure 3). This test object has both random and deterministic parameters and is a system-level model of a midcourse and terminal BMD system. Closed-loop testing has been achieved for this test object. Based upon the success and experiences gained in testing, the extension to more detailed models is in progress. Several key issues have been identified which must be resolved before an effective testing tool for large systems is available. The issues and approaches for their resolution are discussed below.



TRACK AND DISCRIMINATION PROCESSING
INTERCEPTOR ASSIGNMENT
THREAT CLOUD
1 RV TYPE
2 DECOY TYPES
VARIABLE COMPOSITION
THROW WEIGHT CONTRAINT
TRACK AND DATA COLLECTION
DETERMINE ACHIEVABLE DISCRIMINANT
THREAT ACQUISTION
• 7 AVAILABLE DISCRIMINANTS:
• ACHIEVABLE DISCRIMINANT DETERMINED BY (MIPS) x (T)/(NO. OF THREAT OBJECTS)
PLATFORM
SENSORS
DATA PROCESSOR
INTERCEPTORS

Figure 3. System Level Model

The adaptive tester must be able to efficiently locate a point on the boundary in the face of the randomness in the system under test. Some method must be derived to allow an accurate determination of the "position" with respect to the boundary in a single or small number of runs. One approach for keeping the number of tests within reasonable bounds involves using an expected value analog of the random process that will give accurate estimates of the position within threat space. This expected value analog would allow an estimate of location to be used by the PPA in searching for the boundary. Once the expected value analog indicates the boundary has been reached, the stochastic model can then be run a sufficient number of times to achieve the desired confidence that the boundary has been located. For a simple system, the feasibility of construction of an expected value analog which gives good agreement between deterministic and stochastic models (Figure 4) has been demonstrated. The difficulties of doing this for a more complex model are currently being investigated. It is not unreasonable to expect that during a hierarchical software development functional models of the system would be developed, but the requirement that these models reflect to a certain accuracy the stochastic nature of the system may prove too difficult to achieve.

The adaptive tester must be able to accurately characterize the performance boundary. After a sequence of boundary points is obtained, their connectivity is still in question. This is due to the high dimensionality and complexity of the performance surface being defined. Characterization of the performance boundary has been achieved as a polynominal in threat parameters where the coefficients are determined by regression analysis on the boundary points; however, automated approaches investigated so far tend to treat the system as simply connected, and for this reason the cognitive capabilities of the human mind prove invaluable in perceiving the boundary shape.

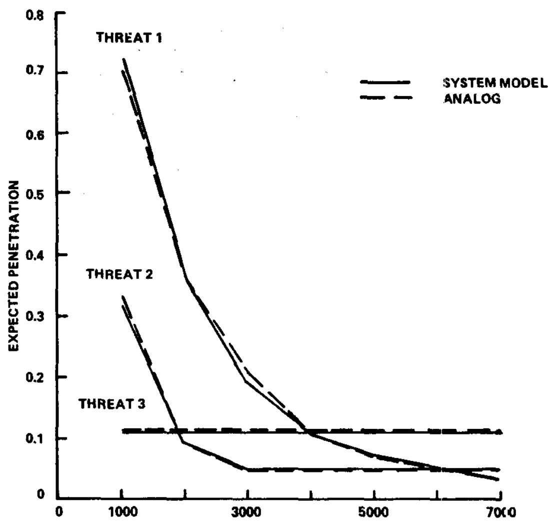


Figure 4. Model Comparison

The adaptive tester must be able to search a highly irregular, discontinuous and highly granular surface. The PPA algorithm must be able to efficiently determine the global maximum of the surface and reach the boundary efficiently. The problem is one of classical optimization theory. Five desirable characteristics of the algorithm have been identified.[12] The algorithm must be

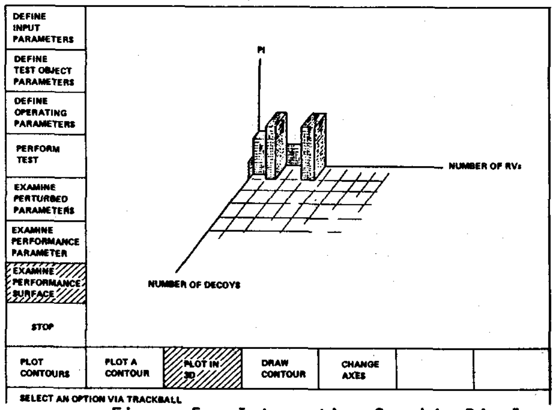
- global--able to find global rather than local extrema
- immune--not sensitive to discontinuities or extreme granularity
- successful--reaches boundary a high percentage of the time
- efficient--reaches the boundary with only a small number of tests
- proximate--a boundary point is reached which is in some neighborhood of the boundary point which is nearest to the initial test point.

The approach taken in the adaptive tester is to combine the highly global and immune characteristics of random search with heuristics to provide an efficient, successful, and proximate approach. The generation of heuristics for this approach requires much insight into the system operation. The heuristics range from the known relationships that exist (e.g., increasing the number of threatening objects will generally increase the expected number of penetrators) to more detailed relations between software parameters at a lower level. The issues are threefold: the time and costs involved in generation of the heuristics, the relationships between heuristics at various levels of detail in system testing, and the transferability of heuristics between systems. Another related issue is the ability to transfer the knowledge of the system embodied in the heuristics to a designer in terms that will facilitate his understanding of the system. Experiences so far have indicated that a relatively complete set of heuristics is required to achieve efficient algorithm performance.[13]

The adaptive tester must also deal with the redundant performance data that is obtained at almost every level of detail. With the large amount of data which is recorded, techniques which recognize significant dependencies among parameters and eliminate redundant parameters before further processing are desirable. It is highly desirable that these techniques be automated and produce those statistically significant parameters which form the basis for the generation of heuristics for a PPA. This has proved to be most difficult to perform in an on-line basis.

Finally, implementation issues also clearly interdict themselves between the successful achievement of an adaptive testing capability. Rapid and efficient communication and modification of the testing system is essential. The user must be able to enter directives required by the tester such as scenario inputs, performance evaluation rules and adaptive algorithm heuristics (Figure 5). The required data which might be of interest for post-processing must also be specified. In addition, there must be a flexible method of threat definition and an input system which can handle system simulations of varying complexity. A large and flexible working data base from which the PPA can extract information and determine the next tests cases must be available. The new cases must also be checked against many constraints.

The theory of the testing of large software systems is such that there are no practical guidelines which guide the testing process. The adaptive tester is a computer-aided approach designed to provide through heuristics a way of obtaining highly desirable information from the testing process. While issues still remain for resolution, the feasibility of such an approach has been demonstrated and, if successful, will have significant impact in lowering testing costs of future system development.


DEFINE INPUT PARAMETERS
DEFINE TEST OBJECT PARAMETERS
DEFINE OPERATING PARAMETERS
PERFORM TEST
EXAMINE PERTURBED PARAMETERS
EXAMINE PERFORMANCE PARAMETER
EXAMINE PERFORMANCE SURFACE
STOP
PI
NUMBER OF RVs
NUMBER OF DECOYS
PLOT CONTOURS
PLOT A CONTOUR
PLOT IN 3D
DRAW CONTOUR
CHANGE AXES
SELECT AN OPTION VIA TRACKBALL
Figure 5. Interactive Graphic Display

The theory of the testing of large software systems is such that there are no practical guidelines which guide the testing process. The adaptive tester is a computer-aided approach designed to provide through heuristics a way of obtaining highly desirable information from the testing process. While issues still remain for resolution, the feasibility of such an approach has been demonstrated and, if successful, will have significant impact in lowering testing costs of future system development.

Software Quality Laboratory

In addition to testing as a means of determining confidence in a program's performance, the application of formal proof techniques provides another equally valuable approach. Much useful information can be derived about the correctness of a program through the addition of redundant information in the form of executable assertions.[14] The goal of the software quality laboratory is to effectively apply the techniques of formal verification to large real-time programs. Even though formal proof techniques for large programs are not practical, incomplete attempts at program proofs provide much enhanced error detection capability. For example, many assertions about a program occur as a natural consequence of the design and can be effectively used in static analysis for common semantic errors.

The software quality laboratory contains tools which assist in detecting common semantic errors, allow instrumentation for testing, and support formal verification of computer programs.
The facility has been designed to analyze both FORTRAN and PASCAL along with two dialects--IFTRAN and verifiable PASCAL. Verification condition generators have been implemented for both languages which handle logical, multidimensional arrays. The simplifier contains many standard simplification rules that can be invoked to cause automatic simplification of verification conditions. Rules not in the simplifier can be applied to individual verification conditions. These can be text replacement or pattern matching rules which can be saved as axioms to be reused. The software quality laboratory executes on the CDC 6400 and CDC 7600 machines and is written in a structured dialect of FORTRAN (IFTRAN).

The approach taken for formal verification using the software quality laboratory is to divide the program into small segments each of which is verified independent of all other parts. Each verification condition is tightly coupled to a verification path through a code segment. This approach is designed to aid the verification of large programs.

The software quality laboratory[15] is most effectively utilized when most of the common syntax errors have been removed by a compiler (Figure 6). Initial checks would be static analyses for common semantic errors. These semantic errors differ from those detected by a language processor in that sequences of statements must be examined rather than a single statement.

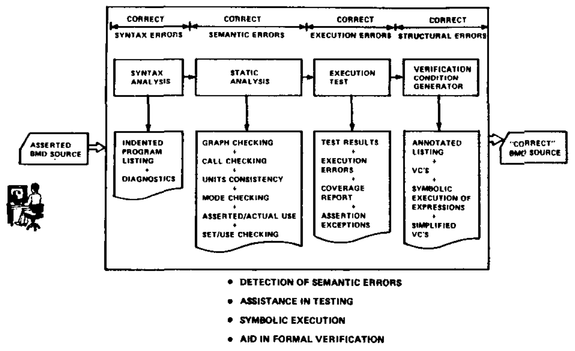

CORRECT
SYNTAX ERRORS
CORRECT
SEMANTIC ERRORS
CORRECT
EXECUTION ERRORS
CORRECT
STRUCTURAL ERRORS
SYNTAX ANALYSIS
STATIC ANALYSIS
EXECUTION TEST
VERIFICATION CONDITION GENERATOR
ASSERTED BMD SOURCE
INDENTED PROGRAM LISTING + DIAGNOSTICS
GRAPH CHECKING + CALL CHECKING + UNITS CONSISTENCY + MODE CHECKING + ASSERTED/ACTUAL USE + SET/USE CHECKING
TEST RESULTS + EXECUTION ERRORS + COVERAGE REPORT + ASSERTION EXCEPTIONS
ANNOTATED LISTING + VC'S + SYMBOLIC EXECUTION OF EXPRESSIONS + SIMPLIFIED VC'S
"CORRECT" BMD SOURCE
• DETECTION OF SEMANTIC ERRORS
• ASSISTANCE IN TESTING
• SYMBOLIC EXECUTION
• AID IN FORMAL VERIFICATION

Figure 6. Software Quality Lab Steps in Validating a Program

These errors are reported in terms of groups of statements that could cause the error rather than a single statement. Without the addition of redundant information, the lab supports static analysis for the following types of errors:

- Set/use errors--variables used prior to setting or initialized and not used. Two types are detected--the use of a variable before it is set and the variable set but not used.
- Mode errors--mistakes with the use of real and integer variables.
- External reference errors--misuse of external routines with incorrect number of parameters.
- Infinite loop errors--control variable not increment on one path.
- Unreachable code errors--structurally unreachable code.

Units consistency checking is provided statically with the addition of assertions about the units of variables. Analyzers compare the right and left hand sides of assignment statements and relational expressions and will flag inconsistencies in units if unlike units are added, subtracted, or compared. Units can be checked across more than one module if each module contains a description of the units for each physical variable it refers to in the form of an assertion.

Another static check is that of asserted versus actual use. Through the use of data flow analysis the actual versus asserted use is checked. This type of checking can be used to detect incorrect usage of variables between modules through the use of an INPUT and OUTPUT assertion of the form INPUT (variable list) and OUTPUT (variable list). An INPUT assertion states that the named variable meets the conditions that it is; it is a global variable, which will have a value when the routine is called, will not be changed in the routine (unless it is also declared as an OUTPUT variable), and is the only global variable used in the routine.

An OUTPUT assertion similarly is used to assert the listed variables are global, will be assigned in the routine, will not be used to supply a variable to the routine (unless also declared as an input variable), and are the only variables set in the routine. This assertion of variables which are set prior to entry to a module (INPUT) and those assigned or read from auxiliary storage (OUTPUT) allows a restriction upon the global variables which may be used or set by a particular module and can also be used during execution to catch errors such as data out of expected range.

Upon completion of the static analysis phase the software quality laboratory supports execution through providing output to show path coverage, false assertions, and values of the input variables upon module entry as well as values of the output variable upon module exit.

Fault detection can be introduced to a program under analysis by the Software Quality Laboratory through the inclusion of assertions with a FAIL clause. While similar results could be obtained with a sequence of IF tests, the assertions provide an explicit statement of conditions under which the code that follows is expected to operate and there is also a separation of error handling from the rest of the code.

Tools to assist in formal verification are also an integral part of the software quality lab. These tools include a verification condition generator and a simplifier, which may be used interactively to verify single modules. The verification condition generator uses assertions which have been inserted into the source code to generate conditions of the form A implies B, where A is the initial assertion on a program path conjuncted with predicates along the path and B is the assertion at the end of the path. The required substitutions are made by symbolically executing the final assertions and any predicates backwards to the initial assertion. Using the INITIAL assertion to state conditions which are true when the module is entered, the FINAL assertion to define conditions which are true upon exit from the module and the ASSERT statement to express loop invariants or any condition that is true at that point, verification conditions are then generated for each verification path in the program. Each logically possible path between program entry, loop entry, loop exit and program exit corresponds to a verification path. Each verification path must begin and end with an INITIAL, FINAL or ASSERT statement. The advantages for associating verification conditions with verification path are described in Reference 5.

The verification condition simplifier uses a set of arithmetic, logical and relational simplifications in an attempt to reduce a verification condition to true. The user is presented the results of the attempt to allow input to supply additional simplification rules which may be unique to the problem under analysis. Once a new rule has been applied, the modified result is sent to the simplifier and a new result presented. To reduce the effort of repeatedly entering rules, a rule is assigned an axiom number and saved in a library of rules.

Interactive simplification using the software quality laboratory is designed to aid in the development of assertions and to improve the performance of the simplifier. The user interacts with the software quality lab through requesting verification conditions, providing trial assertions, specifying additional simplification rules, and requesting the symbolic execution of expressions. The software quality lab responds by generating verification conditions from assertions interactively entered or in the code, simplifies the verification conditions, symbolically executes expressions over specified program paths, validates simplification rules, and applies them to verification conditions.

It is felt that upon completion, the Adaptive Tester and the software quality laboratory can provide cost effective analysis of large real time system. These tools, in the formative stage, hopefully will provide the framework from which more extended capabilities can be developed. The integration of testing into the software development cycle will also provide a big step toward providing a significantly increased capability to produce demonstratably reliable software systems.

Conclusions

Experience and interest in the SDS has been high and several components of SDS are being used with success in a wide variety of applications. These experiences are enhancing our understanding of the potential use of the computer as an aid to software development. Also the adaptive testing and quality assurance programs, for example, show much potential for improving the efficiency of the testing process through computer aids.

The problems in developing software for large systems have been compounded by the increasing hardware capability, and this complexity must be addressed in any software approach. The issues provide the development cycle but have their root in the requirements area. This definition of the requirements for a processing component of a system in such a way as to allow design freedom yet achieve preciseness remains a yet to be achieved goal.

Acknowledgements

Specifics concerning the Adaptive Tester and software quality laboratory are taken from documentation provided by General Research Corporation, Santa Barbara, California, who is developing them under support of the BMD Advanced Technology Center, Huntsville, Alabama.

References

1. C. G. Davis and C. R. Vick, "The Software Development System," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 69-84.

2. C. G. Davis, "Requirements Problems in Large Real Time Systems Development, INFOTECH State-of-the-Art Report, Structured Analysis & Design, 1978.

3. C. R. Vick, "First Generation Software Engineering System," Proceedings ACM National Conference, 1977.

4. C. G. Davis, "Testing Large Scale Real Time Systems," INFOTECH State-of-the-Art Report, to be published.

5. M. W. Alford, "The Software Requirements Engineering Methodology (SREM) at the Age of Two," Proceedings COMPSAC 78, November 1978.

6. R. C. Slegel, "Applying SREM to the Verification and Validation of an Existing Software Requirements Specification," Presented COMPSAC 78, November 1978.

7. C. R. Vick, Research and Development in Computer Technology, "How Do We Follow the Last Act?" Keynote Speech Preceding 1978 International Conference on Parallel Processing, September 1978.

8. E. F. Miller, Jr., "Tutorial Program Testing Technique," IEEE Catalog No. EH0130-5, 1977.

9. C. V. Ramamoorthy, R. C. Cheung, and K. H. Kim, "Reliability Integrity and Large Computer Programs," Electronic Research Laboratory, University of California, Berkeley, Memorandum No. ERL-M430, March 1974.

10. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions Software Engineering, Vol. I, No. 2, June 1975.

11. R. G. Uttley, et al., "Adaptive V&V Midyear Report," General Research Corporation, Santa Barbara, California, CR-6-767, July 1977.

12. D. W. Cooper and R. G. Uttley, "Adaptive V&V Research Plan [Updated June 1977] General Research Corporation, Santa Barbara, California, CR-4-767, August 1977.

13. R. L. Stone, E. R. Buley, R. G. Uttley, "Research Progress Evaluation Report," Final Report, Volume 2, General Research Corporation, Santa Barbara, California, CR-8-9-767, February 1978.

14. S. H. Saib, J. P. Benson, and R. A. Melton, "Executable Assertions--An Aid to Reliable Software," Proceedings of the 11th Annual Asilomar Conference on Circuits, Systems, and Computers, Pacific Grove, California, November 1977

15. S. H. Saib, "A Methodology for Program Verification,"Proceedings of the Summer Computer Simulation Conference, Chicago, July 1977.