LEVEL II (12)

NOSC

AD A056061

NOSC TD 139

NOSC TD 139

(9) Technical Document 139

(6)

ISSUES AND PERSPECTIVES IN THE VALIDATION OF TACTICAL SOFTWARE.

(10) R. N. Goss

(11) RN Goss

1 February 1978

DDC
RECEIVED
JUL 10 1978
F

(12) 25p.

(16) F61101.

(14) NOSC/TD-139

AD No.
DDC FILE COPY

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

78 07 06 059

313 150

JOB

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

RR GAVAZZI, CAPT, USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

The work was performed by the Computer Systems Architecture Branch of the Naval Ocean Systems Center (formerly the Naval Electronics Laboratory Center) under project number F61212 (NELC Z291), program element 62766N.

Released by
RR Eyres
Tactical Computer Systems
Architecture Division

Under authority of
JH Maynard, Head
Command Control-Electronic Warfare
Systems and Technology Department

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC Technical Document 139 (TD 139)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ISSUES AND PERSPECTIVES IN THE VALIDATION OF TACTICAL SOFTWARE		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) RN Goss		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Ocean Systems Center San Diego, CA 92152		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS F61212 (NELC Z291) 62766N
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Ocean Systems Center San Diego, CA 92152		12. REPORT DATE 1 February 1978 13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software engineering
Software validation and verification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document presents the issues and perspectives in the validation of tactical software.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

78 07 06 059

CONTENTS

1.0 INTRODUCTION . . . page 3

2.0 BACKGROUND . . . page 3

3.0 NECESSITY FOR SOFTWARE VERIFICATION . . . 6

4.0 CORRECTNESS: THE FORMAL CRITERION OF QUALITY ASSURANCE . . . 7

5.0 THE PROGRAM GRAPH: A UNIFYING DEVICE . . . 10

6.0 VALIDITY: THE FUNCTIONAL CRITERION FOR QUALITY ASSURANCE . . . 12

7.0 VALIDATION TOOLS . . . 14

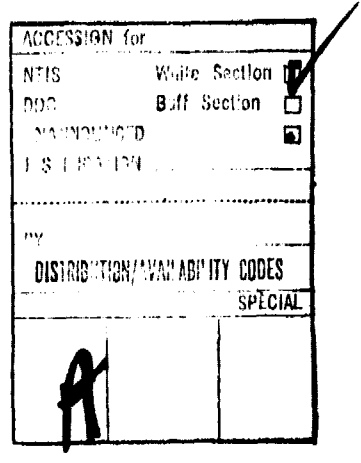
8.0 SOME SPECIFIC TOOLS . . . 16

- A. Data Flow Analysis . . . 16
- B. Symbolic Execution . . . 16
- C. Clock Probes . . . 17
- D. Direct Code . . . 17
- E. Test Generation . . . 17
- F. Standards Enforcement . . . 18
- G. Accuracy Determination . . . 18
- H. Simulation . . . 18

9.0 SCOPE OF THE REQUIREMENTS . . . 19

10.0 SOFTWARE OR HARDWARE? . . . 20

11.0 REFERENCES . . . 22



1.0 INTRODUCTION

This report represents some of the results obtained under Project Z291 of the NOSC In-House Research and Development program. The title of the project, "Command Control Distributed System Design and Validation Processes," suggests that the task embraces two separate efforts, and indeed this is the case. This document deals only with validation processes; the results on system design are reported elsewhere.

The present report is accompanied by a requirements document[1] which it is intended to complement. The latter, constrained to follow a rigidly standardized format,[2] is not a suitable medium for discussing issues that fall outside of its limited range. Many of the issues are relevant to the present and future development of validation tools and to the way those tools are deployed. The documentation standards[2] predate the rise of software validation as a rational discipline and are, on that account, not necessarily well planned to convey to the software system designer the special characteristics validation tools must have vis-a-vis the more usual software products. This report may be considered as an expansion of paragraph 2.1 of reference 1.

Another document of interest and relevance to the readers of the subject matter is reference 3 which is concerned with furnishing a general orientation in the field of software verification rather than with developing requirements. The principal themes of reference 3 are (i) that the activity of producing reliable software must begin with a carefully designed and constructed programming language, especially in its data-handling capabilities, and (ii) that testing is a multifaceted endeavor that demands special expertise and intuition. We agree completely with both of these assessments. They cannot be too strongly emphasized, and their implications for validation tool requirements are clear. No tool, no matter how sophisticated, is going to be completely successful in overcoming programming problems that stem from inept language characteristics. The best tool will be one that the quality assurance tester can adapt to any test strategy. The last point has been a guiding principle in writing the requirements.

This document does not duplicate reference 3, and in some minor points may take issue with it, but there are no incompatibilities. In particular, the other references in this report supplement those of reference 3, which, in addition, contains an extensive bibliography on software reliability.

Some of the remarks made by the author on 1 July 1977 to participants in a UCLA Extension course in Command and Control for Air Force, Army, and Navy systems, are incorporated herein.

2.0 BACKGROUND

The current large general-purpose standard computer for Navy shipboard or shore use is the AN/UYK-7, and the complementary smaller computer is the AN/UYK-20. The Navy has hundreds of each. The standard source programming language is CMS-2, of which CMS-2Y and CMS-2M are the respective versions for the AN/UYK-7 and the AN/UYK-20.

---

[1] NOSC Technical Document 138. "Functional Description of a Validation Tool for CMS-2 Software," by RN Goss, 1 February 1978

[2] Department of Defense, DoD Manual 4120.17-M. "Automated Data System Documentation Standards Manual," pp 2-3 through 2-16, December 1972

[3] NELC Technical Note 2949,* "Software Verification, A State of the Art Report," 12 May 1975

*NELC technical notes are informal publications intended primarily for use within the Center.

PRECEDING PAGE BLANK-NOT FILMED

Another version, the CMS-2Q, is used for the UNIVAC CP-642/CP-642B, CP-901 (1830A), and 1218 computers. Compilers for these versions are supported by the Navy at a number of locations.[4] There is also a Universal CMS-2 Programming Language under development.[5] The functional description envisions a compiler that will be applicable to software written in any version of the CMS-2 language authorized for use.

The problem of software reliability has been a matter of deep concern in recent years. On the one hand, it has proved to be much more difficult to control than earlier Navy policy planners had anticipated, and, on the other hand, it has become an increasingly critical factor in the successful conduct of tactical operations. There is general agreement that if any kind of permanent change for the better is to be effected, it will have to come as a result of a top-to-bottom overhaul of the process by which the tactical software comes into being at the same time. Present investments and commitments make any such wholesale transformation impossible. The alternative is to achieve such reform as we can by tightening up current procedures and exploiting available technology to make the systems now in the Fleet less vulnerable to the forces which degrade their performance. The project reported on here was undertaken in that setting. It does not address the question of making software systems more reliable through better design, but only how to more effectively test the systems that we already have. Its focus is not on change, but on gathering and using information to advantage.

The direction of the effort has shifted somewhat from its original goal, but such a midcourse correction is not unheard of in a project with research overtones. At first the intent was to carry out the complete development of a validation tool for CMS-2 from the establishment of requirements to the final production, the object being to show that it could be done. In the meantime, however, an independently funded contract was let to Science Applications, Incorporated, a firm with considerable expertise in the field, to produce just such a tool as a part of the SDL (Systems Development Laboratory) program.[6] A collateral effort of the original planned development was to have been a study and possible production of a similar tool for the AN/UYK-20 MACRO assembler. But this undertaking, in turn, was also preempted, this time by the McDonnell Douglas Astronautics Company with their Program Evaluator and Tester (PET). In view of the accomplishments of these qualified entrepreneurs, it seems pointless to attempt to duplicate their efforts.

The task of writing a set of requirements has been retained. Having been relieved of the necessity for carrying out the rest of the development process, we have been able to concentrate more on the requirements themselves which is entirely in accord with the currently accepted doctrine that the front end of the software development cycle should receive more careful attention. As a result, a special effort has been made to make certain that the requirements presented (ref 1) are specific, unambiguous, and complete. These requirements go beyond the provisions of the tool being prepared for the SDL, and, if implemented, will equip the quality assurance investigator with a versatile aid for comprehensive and informative software testing. Such a tool would represent the practical limit of the present state-of-the-art in that field.

---

[4] Prokop, Jan, ed, *Computers in the Navy*. Naval Institute Press, 1976

[5] Fleet Combat Direction Systems Support Activity, San Diego. "Specification of the Universal CMS-2 Programming Language," SS-2006, November 1973.

[6] Science Applications, Inc, San Francisco. Report RP-30, "Program Performance Specification Document for Automatic Test Analyzer for CMS-2M (ATA/CMS-2M), by M Wilkes and MR Paige, 11 April 1977

Despite the autonomous character of the functional description,[1] it does not exist in a vacuum. It is a product of its environment and the existing philosophy of software certification. As it happens, we are in a state of transition with respect to such a philosophy. The point of view that has prevailed up to now is best articulated by a quotation of RADM Frank S. Haak (ref 4, p 16):

> The . . . task involves unit testing to ensure that each program operates without error under all possible conditions.

But in recent years the body of opinion has undergone a shift toward a less absolute position, well expressed in reference 7, page 6:

> A software error is present when the software does not do what the user reasonably expects it to do . . . Software errors are not an inherent property of software.

Clearly one's approach to testing will be different depending upon which school of thought one accepts, and, as suggested, the second viewpoint has begun to gain support without supplanting the first (as yet).

We are, in this report and in the requirements, firmly in the second camp and predict that the doctrine of "reasonable expectation" will eventually prevail. Nevertheless, there is a strong residue of support to be found for the belief in "error-free operation under all possible conditions." This is no doubt partly due to the fact that some persons are instinctively uncomfortable with ill-defined situations, but probably for the most part traceable to a concern about the unthinkable consequences of certain malfunctions of military software. Such failures must not be allowed to happen.

It is our belief that the possibility of catastrophic malfunction is a problem that must be specifically addressed by the software designer. Although the critical decision points should be identified for special treatment by the quality assurance investigator, the burden of preventing ruinous failures is not his. He enters the picture at too late a point and is not given the infinite time or resources that would be needed "to ensure that each program operates without error under all possible conditions."

Unfortunately, the flames continue to be fanned under the notion – a misapprehension in our opinion – that errorless software has a fair chance of being realized. The fuel is supplied by the proponents of correctness proofs, who, in their enthusiasm are often less than candid about the severe limitations on the applicability of the concept. It is not irrelevant to point out that the nature of the challenge which the correctness problem offers is ideal for study in an academic setting in which dissertation topics need to be generated. In this instance, it may well be that Kaplan's "law of the instrument" – if you give a small boy a hammer, it will turn out that everything needs pounding – is operative, especially if funds for the pounding can be obtained from a willing source.

Since the correctness approach is rarely discussed with candor by those who have a vested interest in its success, we have taken the liberty of including a brief critique of it in this document. A full understanding of the verification issue, which is what the testing requirements implicitly address, must include a grasp of the role of program proving in the present and foreseeable future development of tactical software.

The last sentence immediately above highlights an unexpected difficulty that was encountered in the course of writing this report: The lack of an adequate and agreed upon vocabulary for distinguishing the key ideas from one another and expressing them with precision. The nickname "V&V" is applied to the general area of concern in which *verification* was at one time supposed to mean the search for errors in a simulated environment and *validation* to mean the search for errors in a real environment. As it has turned out, the

---

[7] Dijkstra, EW, "Programming Methodologies: Their Objectives and Their Nature." In *Structured Programming*, D Bates, ed, pp 203-216. Infotech International Limited, 1976

distinction has not been a useful one and the two words have been appropriated to express other meanings since the necessity for terms that do discriminate among those that need to be kept separate has become more and more acute. The present situation is chaotic. In the absence of a standardized vocabulary, we have been forced to make our own definitions and even to inject a new term, *authentication*, into a picture that is already confused. For this we apologize and declare in extenuation that we have made every effort to be self-consistent in our use of all the words which appear here and in the associated requirements document.

3.0 NECESSITY FOR SOFTWARE VERIFICATION

If, in the days when the computer was emerging as a revolutionary new element in military tactics, any of the people concerned with the production of computer programs foresaw what a towering issue software complexity* was to become, they did not go out of their way to articulate their views. At that time complexity was not thought of as a problem at all. The prevailing wisdom was that, with the machine as an ally, there was virtually no limit to what man could accomplish in dealing with intricate decision patterns. But complexity since then has become a question of great urgency and its effects on military software have been far-reaching and frustrating; they have eventually led to reevaluation of the role that each component in the software development chain carries out and have culminated in directives for tighter and more realistic management controls. Certainly one of the major foci of the new interest in the software process is at the point of quality assurance; it is here that the software faces a moment of truth as it leaves the hands of the producer and, under the scrutiny of an umpire, passes over to carry out its functions at the hands of the consumer. Quality assurance is supposed to make the user of tactical combat systems reasonably secure that the software delivered will not churn out unpleasant surprises when he tries to make it work. That quality assurance up to now has been inadequate is admitted by almost everyone; those who are informed recognize that it is simply a result of the overwhelming cumulative effect that complexity has upon the process of turning out a large computer program, and that it is not necessarily a manifestation of human incompetence.

Two major thrusts have been made to alleviate the problem. One, about which much has been written, aims at improving the programming discipline and technique. While a successful effort here tends to increase programmer productivity and perhaps to reduce harmful side effects resulting from nonuniform coding practices, it has not wholly succeeded in coming to terms with the inherent complexity of a problem itself as distinct from the way it is decomposed for sharing among several programmers. Thus, structured programming (under whatever name) has had important benefits but it has by no means been a cure-all.

The second approach is directed at the program rather than the programmer. Although program simplification, where possible, is certainly one of the goals of this approach, it does not see the problem as one of simplification alone. After all, complexity is not some sort of accidental demon that must be exorcised; it is merely an acknowledged property of the real world. Insofar as computer programs faithfully reflect that real world, they are inevitably going to be complex, and increasingly so as the models of the world become more realistic and comprehensive. The second approach, therefore, sees complicated programs as nothing more than normal phenomena which need to be understood rather than deplored.

---

* "Complexity" is used here only in its general sense and does not refer to the burgeoning technical area of time-memory estimates in terms of program size.

There are two distinct schools of thought, however, when it comes to the practice of quality assurance of complex programs. One might be called the global viewpoint: it regards a program as a logical system which may be verified once and for all by formal procedures analogous to those used in proving mathematical theorems. The objective of quality assurance, according to this view, is to establish the *correctness* of a program – a program being *correct* just when it satisfies the program specifications (ie a mathematical description of the concept that is to be programmed) and free it from implementation errors. The other viewpoint places less emphasis on the structure and more on the functions that a program is to perform. It is not so much interested in logical consistency as in making sure that under particular combinations of conditions certain results will be obtained. Quality assurance is conceived in terms of careful testing to yield information about the performance of a program under conditions specified by the tester. This process is called *validation* and the associated property is *validity*. A program is *valid* when it has successfully passed a battery of tests. Since we need to make a distinction between validation and validity, we will adopt the new term *authentication* to mean the establishment of correctness. *Verification* will be taken as the common term embracing both authentication and validation. Although these terms are not standard (there is, in fact, no universally accepted terminology), they are convenient for making the necessary distinctions and do no violence to the concepts they name.

The concept of correctness has aroused considerable interest recently, with hints that authentication may be an attractive alternative to validation as the main resource for quality assurance. It is not hard to understand why this might be an alluring prospect. Quality assurance, up to now having to depend exclusively on validation, has not made high marks. Unpleasant surprises have been far too frequent. Often the user of naval tactical software has been given programs which have not even met the requirements. Since quality assurance is supposed to catch such things, it becomes a target for criticism for not fulfilling its function; the scapegoat is validation. Hence, when correctness is suggested as a possible alternative to validity, it is eagerly – often uncritically – embraced.

It is essential that correctness be seen in the proper perspective. While it may assume importance at some future time, and for that reason deserves the attention of anyone seriously concerned with the means for producing more reliable software, it has for now been oversold by its proponents as a replacement for the conventional testing approach. The impression that it guarantees elimination of all errors has been cultivated, perhaps due not so much to any intent to mislead as to R&D culture that encourages puffery. At any rate, a discussion of its drawbacks is needed to strike a balance. In the following section, we have intentionally accentuated the negative not to rule out authentication as a weapon in the quality assurance arsenal but rather to alert the prospective software customer who may be intrigued with the idea of formal proof to the actual character of authentication when placed against validation, especially in view of the increasing availability and sophistication of validation tools.

4.0 CORRECTNESS: THE FORMAL CRITERION OF QUALITY ASSURANCE

As long ago as 1960 it was pointed out that programs are algorithms; algorithms are similar to the proofs of theorems in mathematics. By 1960 there had been a great deal of interest by mathematical logicians in discovering the formal criteria that alleged proofs must satisfy in order to be valid; it was only natural that these studies should be extended to include programs as well.

The advantage in having the correctness of a program established is quite clear. If the user knows that, given a certain input (which, of course, must be restricted to some family of legitimate inputs), he is always assured that he will have a corresponding output, he can relax, confident that nothing can go wrong. Moreover, testing, which is time-consuming and expensive, is rendered unnecessary, for the user is *guaranteed* that the program will work correctly.

It is no wonder that correctness is seductive, but one must not fail to look for the fine print. The first fact to be faced is that correctness proofs are not yet in a production status. The favorable examples that have been cited up to now are all experimental. Successful experiments are certainly necessary, but they can by no means be automatically extrapolated to realistic states of affairs.

Second, correctness proofs are *static*: that is, they depend upon the structural properties of the program to which they are applied and not upon their execution. There is no way of taking into account new information produced during program execution. Those who are intrigued by the notion that authenticated programs are infallible should satisfy themselves that the troubles which they expect to be eliminated are in fact derivable from deficiencies in program structure. Program execution often brings us face to face with other realities, such as overflow or questions of precision; these and other machine-dependent phenomena are, as a rule, disclosed only by well-designed test runs and would be immune to detection by static analysis. In rebuttal, the advocates of program proofs answer that when a program to be proved is designed to run on a particular machine, the machine characteristics can be incorporated into the initial conditions, just as the hypothesis of a theorem is constructed to include *all* of the requisite conditions for a valid proof in mathematics. Granting the literal truth of this assertion, it is overstating the case to imply that *all* the essentials could be accounted for without some sort of process equivalent to testing.

Third, apart from the calamities that may befall particular programs on particular machines, the dynamic interplay in tactical software systems introduces a higher order of complexity not apparent in the formal structure. This is sometimes called the "team effect." The ultimate way to overcome complexity is, of course, to try to decompose the complex system into simpler subsystems which can be dealt with individually. Both the proving and the testing approaches rely on this principle. But from experience with testing large programs, we have learned that the whole is often greater than the sum of its parts in the sense that such phenomena as sequencing, for example, may cause interactions among the parts that are not apparent in the structural decomposition. The circuitry of the hardware may have effects which do not show in the program flowcharts. Thus, there are dimensions to the complexity of large systems which do not readily lend themselves to formal analysis.

Fourth, to advocate correctness as a criterion for reliable software is to be in conflict with current military doctrine as expressed in numerous recent reports. The new enlightenment recognizes the primary cause of software failure as attributable not to defects in the software as such but to faulty translation of information: requirements which do not communicate the problem, specification errors due to poor requirements definition, design errors due to poorly understood specifications, and so on. Coding errors, in fact, are only a relatively minor annoyance. The concept of correctness, however, is based on the premise that program errors lie in the software itself. To prove that an algorithm correctly mirrors the specifications and that it always terminates may well confirm the programming while leaving the central predicament unrelieved.

Fifth, the analogy that furnishes the rationale for correctness proofs, namely, the similarity to mathematical proofs, is only superficial; it overlooks important differences between the goals, methods, and attitudes of the mathematician, and the programmer.

These differences are subtle but pivotal. The mathematician begins by attempting to establish a new result he suspects may be deducible from what is already known. If he is successful, fine; if not, he is not averse to shifting his sights to make some essential change in the target result in order to be able to construct a valid proof. The collective function of the mathematical community is to forge an interlocking system of results that can be demonstrated. If it turns out that some of the results are surprising – well, that makes it just that more interesting. The mandate of the programmer, on the other hand, is not to build up an accumulation of correct but curious programs – it is to solve particular problems. He is not at liberty to tinker with the problem in order to convert it to something more tractable. In other words, he needs to be concerned with the "truth" of his hypothesis, whereas to the mathematician, a hypothesis may be merely a convenient tentative conjecture.

Moreover, the mathematician has at his disposal a finely tempered language that has evolved through centuries of trial and error by thousands of workers which is now universally employed and understood. This makes for easy communication and facilitates the interlinking of the entire mathematical literature. Would that the programmer were so fortunately endowed! As an example of what is meant here, consider the arithmetic mean of two numbers. The mathematician simply writes (A + B)/2 and proceeds to use that expression wherever he needs it. In contrast, the programmer must think of it as A + (B-A)/2 and deal with special cases depending on the relative magnitudes of the two components.

Again, the theorems of the mathematician do not typically announce results in terms of special cases but rather in all-encompassing generalities. The most admired and sought-after results are those which apply to the widest classes under the weakest possible assumptions. Elegance consists in making the least resources do the most work. In computing, on the other hand, the situation is reversed. One typically considers what happens to each of a series of alternatives by means of a case-by-case examination. Each possibility must be accounted for individually. Elegant programming is that which lays bare the option structure rather than covering it up. Abstruse compactness, so prized by the mathematician for its ability to sweep up a host of implications in a few symbols, is the bane of good programming. In mathematics, for better or for worse, one carves out a niche for himself by becoming expert in a limited specialty. Producing research has a game-like aspect in which the object is to narrow the number of those who can truly understand one's work; it is an intensely egocentric activity. The successful systems programmer, in contrast, must constantly strive for clarity, must make every detail explicit, must resist every impulse to confer on his programs his own trademark, and must be content with – even take pride in – seeing his own contribution lose its identity as it merges into a larger "whole."

Finally, those who profess to see a panacea in correctness tend to be overly sanguine about the ease with which the sheer magnitude of system software will be tamed. As of today, no program with significantly more than a hundred statements or so has been authenticated. Is it really possible to prove the correctness of a large and intricate system without spawning a huge piece of proof software? And will this not in itself be plagued by as many flaws as the programs being examined? The stock refrain is that we can mechanize the proof process by turning all the drudgery over to the computer. Whether this will ultimately become fact remains to be seen, but in any case it will be years before techniques of program proving are brought to the point where they will be potent.

If the current rhapsodizing about correctness tends to exaggerate the role of authentication, an objective appraisal would not rule it out of a total quality assurance program. As indicated above, it obviates the need for certain types of tests, namely, those which are directed at the structural and logical properties of a program, and in some cases it may be

sufficient by itself. It has been used to detect a flaw in a structured, tested program;[7] conversely, errors have been discovered in programs that were previously proved correct (reference 8, page 319). It would seem, therefore, that authentication should be promoted as a supplement rather than an alternative to validation. For the present, if it encourages language designers to make certain that the constructs they program are amenable to proof and if it influences programmers to think in more formal and precise terms, it will have made an important contribution to better software.

Recently the point has been emphasized that any computer program consists of just three elemental types of executable components: the sequence, the branch, and the loop. The different ways of combining these elements are endless in number, but it is not inconceivable that particular combinations might turn up more frequently than others. It is tempting to try to formulate an analogy with the constitution of complex molecules out of the twenty amino acids and an even smaller number of nucleotides. If such patterns can be identified, they would go a long way toward putting the study of correctness on a systematic basis with a corresponding enhancement of value in quality assurance.

5.0 THE PROGRAM GRAPH: A UNIFYING DEVICE

In the preceding section we discussed in rather brief compass the point of view that computer programs have properties of form from which we can extract information that bears upon whether the program faithfully models the problem it is supposed to solve. The distinction was drawn between the examination of the program on the basis of its formal structure and the observation of its performance by means of tests. We now propose to argue that the gap between these two approaches is not, in principle, as wide as might be imagined.

In the first place, the objectives of both verification techniques are exactly the same: to give the user assurance that the software delivered to him will behave as he would reasonably expect it to. The expectation may originate either from an interpretation of the official requirements and specifications or from a common-sense understanding of the problem being addressed; there should be no essential difference. In either case the defining criterion is not an intrinsic property of the software but is the judgment of the user. That the common objective is nontrivial is well attested by past experience. Everyone is willing to accept the fact that events beyond the control of those in the software development chain or the user himself are bound to happen. To prepare for every imaginable contingency would be out of the question both practically and economically – there would still be the unimaginable events. When "unforeseen" incidents occur, they must be handled in an *ad hoc* manner on the spot – that is what unforeseen implies. What have been referred to as "unpleasant surprises" are, rather, those contingencies which the user, unless he has some sort of morbid obsession with Murphy's law, has every right to expect not to occur because of what the software is supposed to be and do.

The two concepts, authentication and validation, complement each other in their modes of attack on the problem in much the same way as form and function are complementary aspects of any artifact. Since computer programs have formal structures whose properties are readily accessible, it does indeed make sense to study those properties and attempt to develop a coherent theory to account for them. It is, in fact, rather startling to many to learn

[8]Myers, Glenford J, *Software Reliability*, John Wiley & Sons, 1976

that mathematically rigorous procedures are not already in everyday use as a routine strategy for ensuring that programs work.[9]

It is equally plausible for the operational character of programs to be exploited. Programs are seldom created in a vacuum merely as playful intellectual exercises; they practically always are devised as a means for solving some extraneous problem. When one considers how to verify that the problem in question is indeed solved, the first, and probably only, method that comes to mind is testing. Choose a set of test cases, try them out, accept the program if it passes, and fix it if it doesn't.

Both approaches to verification depend for their success on the same fundamental principle: that which comes out is the offspring of what goes in. Correctness proofs by their very nature yield valid conclusions only on the basis of precisely stated hypotheses. It is often tedious to have to discover all the assumptions that have gone into the construction of a program and to make each one explicit. Proof is of necessity specific to those assumptions and custom-made merchandise always comes at a high price. The demands are hardly less stringent in the case of validation by testing, for in order to be sure that a test is furnishing the right answer, one has to know exactly what the program is supposed to do. "What we can't specify we can't verify," Perlis has observed,[10] and the truth of the remark is the same no matter which route we may choose to attain verification.

It is a striking and heretofore not sufficiently emphasized point that the principal tool on which both testing and proving rely is the same -- the program graph. Directed graphs are but one of several ways to schematize the organization of a program -- others being, for example, flowcharts and decision tables -- but they are emerging as the most useful. Whether the programmer first constructs a graph and then writes his program using the graph as a guide or first writes the program and then draws the graph is not at issue here. The point is that to each program there corresponds a linear directed graph,* which makes visible the relationships that exist among the various elements while at the same time suppressing the inessential. Since the human mind seems to work especially well with two-dimensional visual patterns, the graph is a handle by which leverage on the intricate workings of a program may be secured.

One often meets the term "graph theory" these days in connection with programming but there is a potential false impression in the use of this term. The theory of graphs is a flourishing mathematical discipline having come into its own about 20 years ago. It has contributed terminology and a framework for modeling the structures of organizations in many fields, including computer programming. But programming has not drawn deeply from the technical content of graph theory; it is not necessary to be proficient in the theory to deal with the graph of a program. Hence, it is somewhat misleading to suggest that program verification involves graph theory as such.

Be that as it may, the program graph is almost indispensable as a foundation for a correctness inquiry. A *graph* is simply a set of points and lines. A graph is *directed* if all of the points are connected by the lines and each of the lines has a preferred direction. In programming applications the points are usually called *nodes* and the lines *arcs*. We have already

---

[9] Davis, Ruth M, "Evaluation of Computers and Computing." *Science*, vol 195, p 1100, 1977

[10] Perlis, AJ, *SIAM News*, vol 10, no 3, p 5, June 1977

[11] Harary, F, *Graph Theory*, p 10, Addison-Wesley Publishing Co, 1969

*Technically a *pseudograph*, since loops are permitted; cf reference 11. As is common, we use the term "graph" to refer not only to the basic structure but also to certain subgraphs obtained by reduction.

identified the elementary executable components of a program as the sequence, the branch, and the loop. By the *structure* of a program we mean the way the program is built up out of these elementary components. The structure is represented by a directed graph when the arcs correspond to sequences and the nodes to branches. A loop corresponds to an arc whose initial and terminal points coincide. The "flow" of a program is shown by the directions of the arcs. The graph sets in relief the relationships that exist in a program in a way that can be easily comprehended.

It is, thus, fairly natural to conceive of a program proof in terms of a walk through the associated graph with appropriate rules of inference being applied at the nodes. We mention in passing that in addition to the familiar geometrical presentation of a graph, which is of inestimable value to the human being seeking to comprehend the intricacies of a program, there are algebraic representations of graphs that are more suited to the computer. The technical manifestation of a proof is not unlike the analysis of switching circuits by Boolean functions.

The graph is also helpful to the test engineer when a program is submitted for validation. It enables him to design his tests with due regard for the execution paths, so that, in principle, the tests will collectively cover the various execution possibilities. In practice, even with the aid of the program graph, this can be a formidable task. It is here that validation tools discussed more fully below enter the picture. The point to be made here is that the program graph is just as basic to validation as it is to establishing correctness.

Since obtaining the program graph is the first step in the investigation of both validity and correctness, it is a potential basis for a unified treatment. It is important, however, that one keep in mind that validation and authentication are distinct processes based upon different concepts. Each has its own domain of application and its own technique, and, as indicated above, its own conception of the nature of a program.

6.0 VALIDITY: THE FUNCTIONAL CRITERION FOR QUALITY ASSURANCE

We turn now to a more detailed discussion of validation, the verification of a computer program by performance testing. In contrast to authentication by proof, whose outcome is an all-or-nothing result, validation admits various degrees. Except in trivial cases, it will not yield certainty but only probability that the program is without flaw. Consequently, the quality assurance investigator must give careful attention to test design in order to maximize that probability under the time and cost constraints he has to observe. In any case, the validity of a program is always relative to the particular set of tests applied. In weighing validation against authentication, it is clear that for the former the most indispensable human attribute is ingenuity in planning, whereas for the latter it is skill in the manipulation of symbols.

Since validity of a program depends upon its execution, the test engineer must be especially adept at interpreting output that may not be anticipated. Failure of a test can come about in many ways; in particular, it can alert the investigator to side effects invisible to an armchair examination of the program. It must be a part of his planning strategy to afford maximum opportunity for such informative phenomena to surface. If the goal of authentication is an error-free program, the aim of validation might be said to be to expose as many aberrations as possible.

In view of the demands upon the test engineer and the key role that his work plays in the software development cycle, there is a potential inclination to frustration and pessimism on his part. Recent years have seen help for him appearing in two different forms. One is at the front end, where there is now a strong movement to require more diligence than has been exercised in the past. When system software projects are better thought out to begin with, and when care is taken to make the requirements and specifications explicit, unambiguous, complete and consistent, then the test engineer can be more specific in what he looks for. This will relieve him from the kind of pressure that arises from looking but not knowing what he is supposed to discover. It should, moreover, mean that he gets cleaner code from the programmer, who himself will have less second-guessing to do.

The other aid, which is now fast becoming a reality and which is the chief concern of this report, is the availability of validation tools. Validation tools are based on the concept of cooperation between man and machine. The quality assurance investigator has in his mind those aspects of the software in front of him that he would particularly like to check out. At the same time, he wants to be sure that he does not overlook something that might be important. The machine can be of help in several ways: for example, by telling him what portions of the program his tests have missed. In order to enlist the aid of the machine, the test engineer has to have at hand the means to make the computer do his bidding – this will be a piece of software called a *validation tool.*

What makes validation tools feasible is the unclouded schematic organization which the program graph provides. By inserting probes in each arc, for example, we can extract certain desired data every time the execution flow of the program passes through the arc. These data can be stored for later processing. Thus, we see that there are three main constituents of a validation tool: something that determines the graphical structure of the program under examination, something that extracts data while the program is running, and something that processes the data after the program is finished.

Each of the three constituents is a piece of software and as such has time and memory requirements of its own. The first and third are carried out off-line and, therefore, do not contend for the attention of the machine with the program being tested. But the second occupies memory during execution of the program, adding to the normal overhead, so must be contrived with care. The probes mentioned above are transfers of control to some sort of auditing procedure. If all this procedure does is to count the times each probe is activated, the overhead demands are not great. Consequently, most of the validation tools in existence contain a basic package which does just that. The best tools of this kind today require a memory increment of about 25% and slow down the computation by about 25%; these are probably close to the ultimate limits. After the testing process, the program is recompiled with the probes removed.

The merit of any particular tool rests in part, then, on the amount of useful information the post-processor is able to wring out of the data produced by the probe and, to be sure, on the skill of the investigator in interpreting the information for his own purpose. There is an intimate relation between the content of the tests and the way they are executed; one has to know what the tests are supposed to show in order to know what to ask the tool to provide. For that reason, the investigator must have a full range of options from which he can select just what he needs. In any case, the probes must be *noninterfering* – that is, their presence in the instrumented program must not affect the program under test except for the unavoidable real-time degradation due to the extra overhead. The situation may be considered as equivalent to a scientific experiment in which the influence of the observer is to be minimized.

Although the instrumentation of a program normally entails the placing of a probe in each arc, this does not mean that the investigator will attempt to look at every possible path through the program where a *path* is a succession of arcs and their separating nodes. This would not be feasible. It is still up to the tester to make a shrewd selection from the set of all possibilities; with the clever use of the validation tools available to him, he can assert with confidence that the probability of error in the program he has exercised has been reduced to an acceptable threshold.

Developers of validation tools of the type we are discussing report that under the conditions prevailing in the writing of large programs today, it is often impossible – not merely computationally impractical but literally impossible – to test every path through an entire program. Programmer A, for instance, builds into his portion certain protective devices for fault tolerance or error resistance. These devices may well inhibit the flow of the program over paths that would result in the errors to be prevented, not only in Programmer A's portion but in Programmer B's as well. One of the experts in the field (Michael Paige) has estimated that in a large program perhaps 10% of the paths will be untestable.

In the program graph, the sequences and the branches separate each other. A reasonable strategy for the tester is, therefore, to test the initial sequence and to make sure that each branch is executed in all possible ways. If the postexecution summary shows that this strategy has failed to reach all of the sequences, the investigator can write new tests to make up for the deficiency.

In summary, the basic validation tool counts the executions of each statement in a program when a set of test data is applied. The investigator then has two types of output. The first is the test results which have historically defined the role of quality assurance in the software development cycle. The second, provided by the validation tool, is something new: the identification of each statement in the program together with the number of times it was executed. These primary execution data may be supplemented by statistical summaries of various kinds, depending upon the desire of the tester. From the output of the validation tool, the test engineer can see at a glance what parts of the program have not been reached by the tests or, perhaps, what parts have been tested more than necessary. After interpreting the figures, he can decide whether his original tests have met the standards of adequacy for exercising the program or whether further testing is called for.

7.0 VALIDATION TOOLS

Everything said up to now may be considered as an introduction to validation tools – the chief subject of this report. We will now discuss the rationale and capabilities of the tools. Recall that a validation tool is simply a piece of software intended to assist in the evaluation of other software products.

The idea of cooperative interaction between man and computer is an old one that takes many forms. Often it is espoused with a philosophical axe to grind: to play down the uniqueness of man's endowments and to draw the conclusion that eventually the machine will do everything that man can do, only faster. The ideal state will be attained when an entire operation is fully programmed by formal rules. While this extreme view is seldom advocated outright, and would probably be disowned if challenged, its influence is pervasive. Even among those who deal with military tactical systems, to whom of all people the folly

of such a view should by now be self-evident, one can detect vestiges of belief that the computer will ultimately take over the entire conduct of an operation. It is an immediate corollary then that the process of software verification will also be completely automated.

We do not hold that cooperation is some sort of way station on the route to full automation. Symbiosis is, rather, a desideratum to be pursued on its own account. This view is not only highly tenable, it is liberating: it seeks to allow the complementary potentials of both man and machine to be realized at any given time and it frees one from the obsession that somehow he/she has only half a creation as long as man is in the loop at all. For quality assurance it means that the investigator is encouraged to think of each piece of software to be verified as presenting a different problem. He has to discover what that problem is and then, working within the time and cost constraints imposed upon him, to use his knowledge and skill together with the resources of the computer to solve that problem. In short, his is a distinct profession, and as behooves a professional, he is the judge as to what course to follow to achieve the verification goals. It is up to the system management, in turn, to furnish the investigator with the best facilities available to carry out his responsibility. Among those facilities will be validation tools and the means to apply them.

Validation tools are a comparatively recent development in software practice. According to a brief sketch of the subject,[12] the first attempts at dynamic analysis were reported in 1967 by G. Estrin.[13] Since 1967 several projects have resulted in tools of differing quality, most of them designed to operate on programs written in FORTRAN. All have, in common, the purpose of furnishing coverage data. As more experience has been gained, techniques have been refined and, as would be expected, more recent examples show vastly improved performance characteristics. The usability of validation tools is now virtually independent of the programming language to which they are applied, but the technical implementations, of course, differ from one language to another.

What does such a tool actually do, how is it used, and why is it a desirable "thing" to have? The concept has not changed materially over the decade since it first appeared. The object is to relieve the investigator of the nagging worry that his tests may not have sufficiently covered the spectrum of possibilities that a given program offers. When he has to trace through the program for this purpose himself, the labor is likely to be great. The tracing process is, however, a reasonable candidate for automation. By means of probes, the number of times each statement in the program was executed in performing the test is detected and recorded. With this information, the user can discover much more quickly exactly what he has tested and, on the basis of that discovery, decide whether further testing of the program is called for. The tool is indifferent to the content of the tests -- it is only concerned with how the tests have used the statements of the program.

From the execution counts -- the primary data returned by the tool -- other information can be derived. In addition, the formation of the program graph, which is a necessary preliminary to testing with the tool, entails the generation of certain facts about the program as by-products. It is possible, therefore, that the tester may have a considerable amount of information at his disposal in addition to the test results themselves. In paragraph 3.1 of reference 1, we have listed a number of items of potential use to the tester, even though it is not likely that he would want to utilize them all each time he runs a test. The list is not exhaustive.

---

[12] Science Applications, Inc, La Jolla. Report RP-17. "Discussion of Automated Verification Systems: For Inclusion in SDL," May 1976

[13] Estrin, G *et al*, "SNUPTER COMPUTER -- A Computer in Instrumentation Automation." AFIRS Spring Joint Computer Conference, 1967

8.0 SOME SPECIFIC TOOLS

Once a program has been instrumented with probes to count executions, it is natural to ask whether other data could not also be obtained in the same way. The answer is affirmative but with a *big* caution sign. The warning is that every such accretion adds to the overhead and distorts the real-time performance of the test program. Whether or not the quality assurance investigator decides to avail himself of any particular capability will depend upon what he regards as important to know in making a judgment about the software at hand.

A. DATA FLOW ANALYSIS

We have observed that the program graph, or its equivalent, is fundamental to any type of verification. It is the graph that lays out the program structure – the logical relations connecting the various parts of the program. Authentication certainly depends upon it, and validation of the systematic type we are considering also requires it. But it can be exploited for other purposes. The graph is basic to data flow analysis. This term refers to another verification technique in which the behavior of the data used by the program is studied to draw conclusions about the program operation. The input data are tracked as they are converted by the program into intermediate results and ultimately to the final output. The analysis is static in that it does not require actual execution – and in that respect it resembles authentication – but is heuristic rather than algorithmic. It is concerned with such matters as the efficient allocation of storage for all variables during the course of the program, and for that reason it may be regarded primarily as an optimization aid. Its usefulness for verification is that it detects anomalies in the data flow, which in turn are evidence of programming errors. It may have to be resorted to if tests devised for validation do not yield expected results whose cause is not immediately apparent.

The importance of data base analysis is stressed in reference 3, and it is appropriate to comment parenthetically on it at this point. As the authors remark, one must develop and use means for verifying the structure, contents, and access methods of the data bases of software systems. We agree. But at the present time the locus of such an effort would appear to be in the compiler diagnostics instead of in quality assurance. The quality assurance investigator is, of course, not precluded from devising tests to satisfy himself that errors due to faulty data treatment do not occur, but the close relation of data base analysis to language issues and the nature of the analysis itself – checking rather than testing – suggest that this analysis should be carried out at debugging time and not left for quality assurance.

B. SYMBOLIC EXECUTION

Closely related to data flow analysis is symbolic execution. This is a technique for expressing the value of every program variable at each node of the program graph in terms of the input value of that variable without the use of real data. It verges upon authentication – the intermediate and final values of the variables are just the results of the logical action of the program statements on the input values. A complete symbolic execution is tantamount to a program proof. Evidently both symbolic execution and data flow analysis furnish a great deal of information about a program, and, hence, facilitate powerful diagnostics;

by the same token, they require substantial overhead. The cost effectiveness of purchasing and using such tools must be weighed against the more modest demands (and more modest information return) of the validation tool that merely provides execution counts.

C. CLOCK PROBES

If one can insert probes into a program to activate counters, one can also put in probes to read a real-time clock. Now, it is obvious that clock readings obtained in this way would themselves degrade the real-time performance under test, and one might ask why clock readings would be needed at all if real-time performance were not an issue. The paradox disappears, however, when questions about the time sequencing of steps in the execution of a program are important. Clock probes may be indispensable in that event. When these probes have served their purpose for testing, like the other instruments, they are removed from the program.

D. DIRECT CODE

The validation tool ordinarily responds only to source language and does not take notice of direct code instructions when the direct code is not embedded in a statement sequence. If the direct code is infrequent in the programs expected to be submitted for test, it is not cost effective to try to incorporate into the tool the capability of handling the code as it is not a trivial task to accomplish. The few instances of machine code that there may be can be scrutinized directly by the investigator to determine their effect. On the other hand, if the software to be examined is an assembler, and if assembly language is expected to be regularly submitted to quality assurance, there would be good reason to consider a validation tool for the assembler.

Whether to do so is a management decision having ramifications outside the scope of this document. The issues of programming discipline and the machine-independence of the system software are involved. Quality assurance is necessarily bound to whatever cognizant management deems to be in the best interests of the system under consideration.

A validation tool for an assembler differs from the kind we have discussed in the linguistic signals it must be designed to accept. The latter requires symbols to be named, for example. All in all, different techniques are called for and for that reason the requirements we have set forth[1] do not apply to an assembler.

E. TEST GENERATION

One normally regards design of the tests to be within the province of the human member of the man-machine partnership since intuition and judgment play important roles in that activity. However, if the software is to be examined according to some standard scenario or when the test data are to be chosen from some existing library of data formats, it makes sense to consider programmed generation of the tests by the computer. In such a case a single tool would generate the test input, check the output against expected results, and provide the investigator with a summary printout of the whole story.

F. STANDARDS ENFORCEMENT

Since the production of software for a tactical system is a team effort, the programmers must be held to standards set by project management and higher authority. For better or worse, the quality assurance group is sometimes designated to monitor the observance of those standards. In that event, means will be needed to inform the tester whether prescribed standards have been adhered to. Since comparison of programs under test with formal criteria is routine and repetitive, a tool for carrying it out by machine is indicated. The tool might even be designed to cast the source program into an acceptable standard form.

It should be pointed out that standards enforcement is distinct from either validation or authentication, though not independent of them. It is no doubt more satisfactory in principle to carry out standards enforcement at some stage other than quality assurance. In many cases, such checking is done by the programmer at debugging time or it may be built into the compiler diagnostics.

G. ACCURACY DETERMINATION

When numerical computations are an integral part of a system, especially when the results of computation are passed from program to program, the problem of accuracy can be formidable. Errors due to round off or overflow can accumulate with unpleasant, even disastrous, effects. The control of those errors is properly the job of programmers trained in numerical analysis. However, the quality assurance investigator may desire or be required to determine whether the program variables survive computation with requisite accuracy. Once again, this activity is a reasonable candidate for automation. The method for monitoring accuracy requires that the variables of interest be flagged and probes inserted after arithmetic computations on those variables; we, thus, have a different set of probes from those that indicate execution counts.

H. SIMULATION

Programs are always written to operate within a certain environment and the seal of quality assurance implies that the approved program is certified for the conditions under which it is expected to be actually used. In some cases, the environment is summed up in the indicated allowable ranges of the variables of the program. In other cases, there may be real-time constraints that are not apparent from an examination of the source program itself. Whatever the circumstances that constitute the environment, it is of prime importance to the test engineer to be able to say not only that the program operates properly but that it will do so when subjected to the conditions it will encounter in tactical situations. Unfortunately, it is rarely possible to make the tests *in situ*. An alternative is to create a setting as nearly like the actual environment as possible. In such a case, the software providing the simulated environment may be considered a validation tool.

These brief statements about validation tools are intended to illustrate the opportunities within reach. It is apparent that the possibilities are varied and that collectively they can place a great deal of capacity in the hands of those who must certify the software. Since their application is not limited to a single program but has continuing use even from

system to system, an open-ended library of such tools would seem to be a good investment. At the same time, it should be emphasized that, like any other tool, a validation tool is an inert instrument, realizing its intended function only when it is used properly by a skilled technician.

It is appropriate to mention in passing what a validation tool is not. In the first place, it does not replace debugging. The latter continues to be the responsibility of the programmer and must be exercised with no diminution of vigilance. In fact, it is stipulated that the source program to be validated must be compilable – that is, free from syntactical errors – before the tool can be applied. This is not to say that the tool will not turn up an occasional bug as a by-product, but the major concern is with program content rather than form. Nonetheless, there is an overlap if not in objective at least in technique, and it is not surprising that if the programmer develops a comprehensive diagnostic package, he is likely to have anticipated quality assurance in some of his own tests.

It would, moreover, seem that whenever it is possible for a validation tool to make use of procedures designed for debugging and already available, it would make sense to do so. At the present time, however, it is often difficult for quality assurance to gain access to the data in a form that it might use to advantage. As a result, there is undoubtedly duplication of effort. The answer is that when complete top-down design of software systems comes into its own, the diagnostic and validation tools may be integrated so that they can work together. Looking even further ahead, one can predict, or at least urge, the unification of debugging, optimization, proving, and testing at a single facility in the software development cycle which the programmer can invoke at any time. Such a concept (as the author learned in discussions with the people involved) underlies the research in new tool technology at Harvard.

Verification is not to be confused with field maintenance and repair of the software, which under conditions that are now foreseeable, will always be something the user will have to contend with. However, as changes in the software are made in the field, the question of whether previous certification continues to be in full force may come up. Since it may be impractical to get the software recertified by quality assurance, the field maintenance crew should themselves have a rudimentary validation tool or two to enable them to make rough checks.

9.0 SCOPE OF THE REQUIREMENTS

As indicated, the requirements are supposed to be fairly general statements about the software to be produced. In the case of tactical software systems, they relate the system to a particular mission the Navy is to carry out. The ultimate criterion the software must meet is to satisfy the requirements. As the software development process is currently perceived, the requirements are followed by the design specifications. These specifications, which translate the requirements into the details that govern the writing of the programs, invoke programming expertise in the high-level source language to be used in the system.

Inasmuch as the process of constructing a validation tool is being carried to completion with the production of the Automatic Test Analyzer (ATA) by Science Applications, Incorporated, it may be asked whether preparing a functional description of a validation tool for CMS-2 software[1] is not superfluous, an *ex post facto* creation. The answer is negative.

The Automatic Test Analyzer is one of a sequence of tools representing increasing experience on the part of a group of experts who have been interested in the automatic validation problem.[13] The point of view from which ATA is being produced is that the lessons learned in developing its predecessors are to be applied to make ATA the most efficient tool that the group has yet turned out; in other words, ATA is primarily the result of an internal acquisition of proficiency rather than a conscious attempt to fashion a product to meet externally generated specifications. Consequently, ATA does not go as far as the requirements in reference 1. This is not to imply by any means that the contractor has been deaf to suggestions by the prospective user.

The provisions in reference 1 are intended to eventuate in a very versatile tool that will enable the quality assurance phase of software development for CMS-2 to make definitive judgments. They set out requirements that apply to CMS-2Y and CMS-2Q as well as to CMS-2M and go beyond the capabilities of any tool now in existence. However, they are within the present state of the art. They do not intentionally favor the way of thinking or the expertise of any particular contractor. A principal objective is to make each capability optional with the user of the tool so that testing may be specifically adapted to the idiosyncrasies of each program presented.

The question of the higher-order language in which the validation tool is to be written, as distinct from the language (CMS-2) of the programs subject to test, has been left open. It is not sufficiently critical to warrant a special requirement, thereby possibly limiting interest on the part of some potential contractors. Since most of the validation tools extant are written in FORTRAN, the absence of such a stipulation will most likely mean, other things being equal, that ANSI FORTRAN becomes the operative language. The use of FORTRAN is alleged to give the tool portability. But the door is not to be closed on other possibilities; the belief here is that the requirements should be concerned with the product and not the means of production.

Tools for the assemblers for the AN/UYK-7, AN/UYK-20, or other processors accepting CMS-2 have not been included. Although the wording of the requirements for a validation tool for assembly-language programs would not be unlike that of reference 1, the techniques for constructing a tool for machine-dependent software are sufficiently different to preclude treatment of the former as a variant of the same task. The machine-dependent case is to be regarded as the next step after tools for higher-order languages. In this connection, we once again call attention to the PET Program.

10.0 SOFTWARE OR HARDWARE?

Assuming that the decision has been made in principle to invest in validation tools in order to enhance the effectiveness of quality assurance, the further course is fraught with choices involving the usual tradeoffs. For example, the estimated usefulness of a particular tool for the kinds of programs to be tested must be set against its cost. Somewhat less obvious is the option which recent technology offers: should the tool be implemented in software, firmware, or hardware?

That this is a live option may not be apparent at once, for we have been conditioned to perceive hardware and software as occupying separate realms, each with its own well-defined function. The distinction is becoming increasingly blurred, however, and changing

cost ratios now make the allocation of function no longer something established *a priori*. The decision today may not be the one for tomorrow. Although software is still to be preferred for the majority of tools because the nature of their application demands flexibility, it is possible that specially designed hardware modules might be recommended under certain conditions. For example, if the memory in the central processor available for verification were not sufficient, an auxiliary device might be called for and, having gone that far, one might find it reasonable to include in the circuitry of that device some of the repetitive operations of the tool. Again, when one considers automating any part of the verification process in order to utilize the man-machine dichotomy effectively, the question of hardware should be a part of that consideration.

There also is an element of psychological value in being able to reassure the user that the programs being turned over to him have been certified by machine. In the first place, automation confers a kind of assurance that the process is well understood. In the second place, hardware malfunction is usually due to random failure of some physical component which can be located with precision and then fixed by a mechanic or engineer. The unreliability of software, in contrast, is often due to defects of design and consequently has a built-in quality. There is a lingering suspicion that the fixing process may introduce errors of its own. Therefore, when the validation is itself performed by software, it is not surprising that less credibility may be attached to the result.

The ultimate goal is to supply the quality assurance investigator with a set of functions from which he can construct verification procedures for any software that reaches his desk. Some of these functions may be preprogrammed hardware units, and others software routines. As microprogramming techniques give more control of the hardware elements to the programmer, the opportunity for multiform solutions precisely tailored to individual testing situations will increase.

This is the way the picture looks today. But the course of events often mocks the seemingly most well-founded predictions. Even now on the horizon there is the expectation of hardware that directly executes higher-order languages. Adoption of this type of architecture would place in much bolder relief the relation of the output program to the original system requirements by eliminating the cumbersome apparatus of compilers, assemblers, and complicated operating systems and would allow the quality assurance investigator to render his endorsements with greatly increased authority.

11.0 REFERENCES

1. NOSC Technical Document 138, "Functional Description of a Validation Tool for CMS-2 Software," by RN Goss, 1 February 1978

2. Department of Defense, DoD Manual 4120.17-M. "Automated Data System Documentation Standards Manual," pp 2-3 through 2-16, December 1972

3. NELC Technical Note 2949, "Software Verification: A State of the Art Report," 12 May 1975 *

4. Prokop, Jan, ed, *Computers in the Navy.* Naval Institute Press, 1976

5. Fleet Combat Direction Systems Support Activity, San Diego. "Specification of the Universal CMS-2 Programming Language," SS-2006, November 1973

6. Science Applications, Inc, San Francisco. Report RP-30, "Program Performance Specification Document for Automatic Test Analyzer for CMS-2M (ATA/CMS-2M), by M Wilkes and MR Paige, 11 April 1977

7. Dijkstra, EW, "Programming Methodologies: Their Objectives and Their Nature." In *Structured Programming,* D Bates, ed, pp 203-216, Infotech International Limited, 1976

8. Myers, Glenford J, *Software Reliability.* John Wiley & Sons, 1976

9. Davis, Ruth M, "Evaluation of Computers and Computing." *Science,* vol 195, p 1100, 1977

10. Perlis, AJ, *SIAM News,* vol 10, no 3, p 5, June 1977

11. Harary, F, *Graph Theory,* p 10, Addison-Wesley Publishing Co, 1969

12. Science Applications, Inc, La Jolla, Report RP-17. "Discussion of Automated Verification Systems: For Inclusion in SDL," May 1976

13. Estrin, G *et al,* "SNUPTER COMPUTER – A Computer in Instrumentation Automation." AFIRS Spring Joint Computer Conference, 1967

* NELC technical notes are informal publications intended primarily for use within the Center.