## General Disclaimer

### One or more of the Following Statements may affect this Document

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

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

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

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

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

Produced by NASA STI Compliance and Distribution Services

77-1385

# A METHODOLOGY FOR TESTING COMPUTER PROGRAMS[1]

Leon J. Osterweil
Boeing Computer Services
Space and Military Applications Division
Seattle, Washington 98124

## Abstract

This paper advances and explores the idea of combining into a single methodology the validation and verification techniques - formal proof, test path generation. dynamic testing, symbolic execution and static analysis. The characteristics of each of the techniques are presented and analyzed. Then it is shown that the strengths and weaknesses of the techniques are largely complementary and represent, from a number of points of view, a natural progression of diagnostic capabilities. A system exploiting this progression is proposed, and some difficulties to be expected in implementing such a system are explored.

## I. Introduction

There has been considerable interest lately in methodologies for the production of high quality computer software. Work in this area has been carried out by researchers in a wide variety of disciplines and covers an impressive spectrum of approaches. Some of the more active current lines of research include: software management techniques [1, 2]; creation of error resistant programming techniques [3, 4, 5]; and design of error resistant programming languages [6,7].

There has also been considerable activity in the creation of program testing verification and analysis techniques. The work in this area has been directed primarily towards two different but related goals -- the detection and examination of errors present in a program, and the determination that a given program has no errors of some particular type. Among the diverse activities in this area, this paper shall focus on four of the major approaches -- namely dynamic testing, symbolic execution, formal verification and static analysis. In this paper, the different patterns of strengths, weaknesses and applications of these approaches will be shown. It will, moreover, be demonstrated that these patterns are in many ways complementary, offering the hope that they can be coordinated and unified into a single comprehensive program testing and verification system capable of performing a diverse and useful variety of error detection and verification functions. This paper also explores some of the problems involved in integrating the methodologies with one another, indicating areas where research remains to be done.

[1] Much of the original work for this paper was done while Dr. Osterweil was on the faculty of the Department of Computer Sciences, University of Colorado, Boulder, Colorado.

## II. Four Error Detection And Verification Techniques

In this section, the different characteristics of dynamic testing, symbolic execution, formal verification and static analysis shall be examined.

### A. Dynamic Testing

The term dynamic testing as used here, is intended to describe most of the systems known as execution monitors, software monitors, and dynamic debugging systems [8, 9, 10, 11]. The term dynamic testing is used because in contemporary usage it has come to suggest the most important feature of this methodology.

In dynamic testing systems, a comprehensive record of a single execution of the program is built. This record -- the execution history -- is usually obtained by instrumenting the source program with code whose purpose is to capture information about the progress of the execution. Most such systems implant monitoring code after each statement of the program. This code captures such information as the number of the statement just executed, the names of those variables whose values had been altered by executing the statement, the new values of these variables, and the outcome of any tests performed by the statement. The execution history is saved in a file so that after the execution terminates it can be perused by the tester. This perusal is usually facilitated by the production of summary tables and statistics such as statement execution frequency histograms, and variable evolution trees.

Despite the existence of such tables and statistics, it is often quite difficult for a human tester to detect the source or even the presence of errors in the execution. Hence, many dynamic testing systems also monitor each statement execution checking for such error conditions as division by zero and out-of-bounds array references. The monitors implanted are usually programmed to automatically issue error messages immediately upon detecting such conditions in order to avoid having the errors concealed by the bulk of a large execution history.

Some systems [9, 10] even allow the tester to create his own monitors, direct their implantation anywhere within the program, and specify where and how their messages are to be displayed.

The greatest power of these systems is derived from the possibility of using them to determine whether a program execution is proceeding as intended. The intent of the program is captured by sets of assertions about the

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

desired and/or correct relation between values of program variables. These assertions may be specified to be of local or global validity. The dynamic testing system creates and places monitors as necessary to determine whether the program is behaving in accordance with asserted intent as execution proceeds.

The previous paragraphs should make it clear that dynamic testing systems have strong error detection and exploration capabilities. They excel at detecting errors during the execution of a program, and also at tracing these errors to their sources. These systems are capable of examining only a single execution of a program, however, and the results obtained are not applicable to any other execution of the program. Hence, the non-occurrence of errors in a given execution does not guarantee their absence in the program itself. Thus, it is seen that dynamic testing systems have no inherent capabilities for demonstrating the absence of errors.

It should also be clear from the previous paragraphs, that the user of a dynamic testing system has access to a wealth of very detailed information about the program he is examining. It is this very precise information that is responsible for much of the power of this approach. It should be observed, however, that this information is obtained only as a result of an execution occurring in response to actual program input data. The generation of this input data is the responsibility of the tester, and in many cases involves quite a significant amount of effort and insight into the program. It is important to recall in this context, moreover, that a significant amount of human involvement is also required in order to effectively use the capabilities of the system to detect and explore execution errors.

In summary, dynamic testing systems provide strong error recognition and exploration capabilities, but are unable to determine the absence of errors. Their results are narrowly applicable, being valid only for a single program execution. These results are quite extensive and detailed, however, providing sufficient material for deep insight into the program. These systems allow extensive human interaction, and their power is most fully realized when a skilled human tester is using them interactively. They require as input a complete set of actual program input data. The success of a dynamic testing run as a vehicle for discovering and exploring errors is largely dependent upon the selection of revealing and provocative input data. This usually presumes the involvement of a human tester who is knowledgeable about the program being tested.

## B. Symbolic Execution

In symbolic execution, symbolic representations (in the form of formulas) are kept for the evolving values of variables instead of numeric quantities. For a given path through the program, the values of all the variables encountered are maintained as formulas. The only unknowns in these formulas are the input values to the program (these may be arguments, in the case when a procedure is being tested, or read-in values); all other values of variables are functions of constants and these input values and, therefore, can be removed by substitution. The formulas can be examined by a human tester to see whether they embody the intent of the program. If so, then the tester has determined that the program will yield the desired results for all executions which follow the given program path. A number of symbolic execution systems have been produced [12, 13, 14, 15].

Clarke's system [12] is perhaps the most interesting symbolic execution system in the context of this paper, in that it indicates better than the others the range of error detection and verification capabilities possible with the symbolic execution approach. In Clarke's system, the execution path which is specified as input is used to dictate the required outcome of all conditional tests along the path. Hence, the path dictates a set of constraints which must be satisfied in order for execution to proceed along the given path. These constraints are in terms of current values of program variables, but through the use of symbolic execution, they can more profitably be expressed as relations in terms of current values of program variables. The system of relations obtained in this way is taken to be a set of simultaneous constraints, and is examined by Clarke's system for consistency. A solution to a consistent set of constraints is a set of values which, when taken as input to the program, will force execution of the given path. If the constraints are inconsistent, then the path is unexecutable -- that is, there exists no data which will effect the execution of the given path.

Clarke's system also creates additional, temporary constraints for the purpose of error detection and verification. For example, whenever a division operation is encountered in the process of symbolically executing a path, a new, temporary constraint is created, constraining the divisor to be zero. The current system of constraints, accumulated to this point, is augmented by this new zero divisor constraint. If this augmented system of constraints is inconsistent, then a division by zero error is impossible along the given path to this point. Otherwise, Clarke's system will attempt to solve the system of constraints to produce program input data which forces the traversal of the given input path, followed by a zero-divide error at the given point. In a similar fashion, constraints are created which test for the possibility of array bounds violations and DO statement loop control variable errors (e.g. non-positive values for these variables).

From the previous paragraphs, it can be seen that symbolic execution systems are capable both of error detection and of a limited type of verification. Error detection for program computations can be obtained by examining the formulas generated by the system. Computations involving the use of uninitialized variables are, moreover, readily detected automatically when the system attempts to create a symbolic

value for the result of the computation and discovers that there is no symbolic value for one of the variables used by the computation. Examination of the systems of constraints arising from the symbolic execution yields other powerful error detection capabilities which have some verification characteristics. If a system of constraints leading up to a division is consistent, yet the system augmented by a zero-divisor constraint is inconsistent, then there is no way that a division by zero can occur at that division, provided the division is reached by the given path. Hence, this procedure is capable of demonstrating the absence of division by zero errors along the path. We shall refer to this as pathwise verification. Clearly pathwise verification is not totally satisfactory because an error such as division by zero may still occur in the program simply by reaching a division along some path other than the one given as input to the symbolic execution system.

The previous discussion also shows that the information gathered about a program by a symbolic execution system is less detailed, but has more general applicability than the information obtained from a dynamic testing system. Virtually no information about specific values of program variables is obtainable from symbolic execution systems. Instead, what is obtained is information about the relations of possible values to each other, and about the manner in which the values are derived. This relation and derivation information is applicable to the class of all executions which follow the given input path. It should be noted that this situation is in marked contrast to the situation for dynamic testing systems. In dynamic testing highly detailed, specific information is obtained, which is, nevertheless, very limited in its applicability.

The human tester plays an important role in exploiting the error detection capabilities of symbolic execution systems, although this role seems to be more straightforward than it is in dynamic testing. The human must examine the formulas produced by the system to determine whether they represent the desired computations. This examination process can be automated to some extent by incorporating an assertion writing capability into the symbolic execution system as is done in EFFIGY [14]. Here the tester asserts what the desired formulas are to be before symbolic execution begins (much as he might create monitors for a dynamic testing system) and has the system verify that the formulas produced are the expected ones. This capability serves to alter the nature of human interaction required from post-execution examination to pre-execution hypothesis. It does not, however, diminish the importance of the human interaction. The pathwise verification capability, in contrast, requires no human interaction. Finally, it must be noted here that the human tester is called upon to supply a program test path to the symbolic execution system, just as he is required to supply program input data to a dynamic analysis system. Moreover, the success of the symbolic execution run in uncovering errors depends crucially upon the selection of an appropriate path--analogously to the situation for dynamic testing.

In summary, it has been shown that symbolic execution systems provide strong error detection capabilities and some pathwise verification capabilities, which fall short of the power of full verification. Symbolic execution systems provide diagnostic information which is applicable to classes of executions rather than a single execution. This is achieved by supplying symbolic relationships between program values in place of precise numeric data. These systems require human intervention and evaluation in order to carry out error detection, although the pathwise validation capabilities require no human assistance. Symbolic execution systems require that a test path through the program be supplied. It is important that the path given be revealing and provocative, thus requiring the skills of a knowledgeable human tester.

## C. Static Analysis

In static analysis systems, the text of a source program is examined in an attempt to determine whether the program is defective due to local malformations, improper combinations of program events, or improper sequences of program events. In order to make this determination, each statement of the program is represented by a small, carefully selected set of characteristics. The static analysis system can then examine each characteristic set on a statement-by-statement basis for malformations, and various combinations and sequences of statements on a characteristic-by-characteristic basis for faulty program structure or coordination. No attempt is made at replicating the entire behavior or functioning of the program. Rather, static analysis attempts to examine the behavior of the entire program only with respect to certain selected features.

The syntax checking of individual statements of a program provides a good example of static analysis. Here each statement is represented only by its source text, or the derived token string. This representation is far from a complete characterization of the statement (its semantics are totally unrepresented, for example), but it is sufficient for the determination of the statement's syntactic correctness. More interesting and valuable error detection is obtained by examining the characteristics of combinations of statements. For example, illegal combinations of types can be detected by examining declaration statements and then examining the executable statements which refer to the variables named in the declarations. Similarly, mismatches between argument lists and parameter lists associated with the invocation of procedures or subroutines can also be made by static analysis systems. In such cases, the invocation and procedure definition statements are both represented by such information as the number of arguments or parameters, the type of each, any dimensionality information associated with each, and input/output characterizations for each. The static analysis consists of comparing the characteristics of the corresponding arguments

and parameters. Mismatched lengths, types and functional usages can be detected in this way. Some of the types of static analysis discussed above are available with most compilers. Other types, such as argument/parameter list agreement are far less common in compilers, but are found in such stand-alone static analysis systems as FACES [16] and RXVP [17].

The use of static analysis techniques to examine sequences of program events enables the detection of still other types of program errors. In DAVE [18] each statement of a program is represented by two lists -- a list of all variables used to supply values as inputs to the computation, and a list of all variables used to carry away values produced as output by the computation. The static analysis then examines sequences of statement executions which are possible given a program's control flow structure, and determines such things as whether it is possible to reference an uninitialized or otherwise undefined variable, and whether it is possible to compute a value for a variable and then never refer to the computed value. In such cases, the static analyzer determines and outputs the statement sequence for which the anomalous pattern of references and definitions occurs. Similarly, it would be possible to scan programs for other improper sequences of events such as openings, writings, and closings of files; and enablings and disablings of interrupts. Paths along which these sequences could occur would then also be determined. It should be emphasized here that the most recent static analysis systems which examine event sequences for improprieties employ search techniques which enable the examination of all sequences of statement executions which are possible, given the flow of control structure of the program. These search techniques, first studied in connection with program optimization [19, 20, 21, 22] are also quite efficient. Unfortunately, the most efficient of them will merely detect the existence of such improper sequences. Somewhat less efficient algorithms are needed in order to determine the actual sequences.

It can be seen from the preceding paragraphs that static analysis systems offer a limited amount of error detection, but are capable of performing certain verification functions. This is done by discarding a great deal of detailed information about the program, retaining only complete information about some specific aspect of the program. The static analyzer then determines the possible effects of executing all program paths considering only those aspects of the execution for which information has been gathered. Hence, static analysis only examines a few narrow aspects of a program's execution, but the results of this analysis are comprehensive and broadly applicable to all possible executions of the program. Here, as in the case of symbolic execution, it is seen that the verification capabilities are obtained without the need for human interaction. A human tester is required, however, in order to interpret the results of the analysis and pinpoint errors. It is important to note that static analysis rarely detects errors at all, but rather detects symptoms of errors, leaving the actual error detection to human analysts. Finally, it is important to observe that static analysis requires no input from a human tester. As output, it produces either paths along which anomalous program behavior is possible, or validation results indicating that no anomaly bearing paths exist.

## D. Formal Verification

In formal verification, the code comprising a program is compared to the total intent of the program, as captured and expressed in the form of assertions. Assertions are used to describe the program output expected in response to specified program inputs. The goal of the formal verification is to prove a theorem stating that the program code actually achieves this asserted input/output transformation. The proof of this theorem is reduced to the proof of a coordinated set of lemmas. The statements of these lemmas are derived from a set of intermediate assertions positioned in specific locations throughout the program code. These assertions describe precisely the desired status of program computations at the locations of the assertions. Differences in status between assertion sets separated in position by a body of code embody the transformation which that code segment is intended to perform. Proving that the code segment achieves the transformation establishes the lemma that the segment is correct. The intermediate assertions must be positioned so that they partition each possible program path into a set of simple (i.e. loop-free) segments. If this is done, then proving all resulting lemmas establishes the theorem that all computation sequences perform as specified. A total formal verification is achieved if the program is also proven to always terminate.

It is quite significant to observe that symbolic execution is the technique used to determine the transformation effected by a given code segment. Hence, the symbolic execution technique is central to formal verification. Formal verification can, in fact, be viewed as a formalized framework for carrying out a rigorously complete and coordinated set of symbolic executions and comparisons to intended behavior.

The great strength of formal verification is this rigor and completeness. The end result of a successful formal verification is a proof that the program completely and correctly implements its intent as captured by the assertions. This result is clearly stronger and more encompassing that any that can be achieved through static analysis or dynamic testing. As observed, it is the most which can be achieved with symbolic execution.

The weaknesses of formal verification are its impracticality and unreasonable underlying assumptions. The impracticality is attributable to the formidable amount of work involved in formal verification. The assertion sets required frequently are numerous and their aggregate

size is often larger than the code they describe. This gives rise to very large numbers of lemmas and proofs. The proofs of the lemmas are often quite straightforward, but not infrequently they are tricky and intricate. As a result the entire verification is invariably far larger than the original code, and is shot through with mathematical intricacy. Thus formal verification requires a sizable amount of work by a person with a good deal of mathematical facility. This occasions great cost.

The size and intricacy of the proof also invites suspicion about the possibility of error. It is unreasonable to assume that a long and sometimes tricky mathematical proof is more accurate than a shorter body of code. In addition, the expected size of the proof often causes the prover to shorten the proof by omitting details. This omission of details further invites errors.

Most of these worries about the reliability of formal proofs can be removed by employing an automated verification system such as GYPSY [23]. Such systems rely upon humans primarily for guidance in producing thoroughly detailed, totally justified First Order Predicate Calculus lemmas in support of the formal verification. Despite the size and intricacy of such proofs, a high degree of confidence in their correctness is justified.

Other unreasonable assumptions underlying formal verification remain, however. The most basic is the assumption that the assertion sets correctly capture the intended behavior of the program. Formal verification can only show that there is agreement between asserted intent and actual code. If the intent is incorrectly captured by assertions the proof cannot establish total correctness of the program in an absolute sense.

More technical proof difficulties cause the imposition of other unreasonable assumptions upon the formal verification process. It is known, for example, that attempted proof of an incorrect lemma may never reduce to an obvious absurdity. Hence, a proof of the incorrectness of an incorrect program cannot necessarily be expected. This is particularly damaging to attempts to use automatic verification systems.

A more serious problem is that formal verification must rely upon an assumed mathematical model of a program's behavior. The purpose of symbolically executing a sequence of code is to determine whether it performs the transformation specified by its bounding assertion sets. The assertion sets are specified in precise mathematical notation. Hence the effect of executing code must be described mathematically. Many commonly used programming constructs, such as GO TO statements and manipulations on floating point numbers and pointer variables, cannot be readily modeled mathematically. Thus the range of applicability of the formal verification technique is reduced.

In summary, formal verification is the most rigorous, thorough and powerful of the four techniques presented here. There are sizable problems in carrying it out, however. The size and intricacy of the work make it costly. The need for exact mathematical models of the desired and actual behavior of a program invite errors and weakening inaccurate assumptions. It is generally agreed, however, that the discipline and deep perception needed to undertake formal verification are useful in themselves. Anticipation of formal verification seems to foster good program organization and design. Attempts at formal verification invariably lead to improved insight into both the goals and implementation of a program.

## III. An Integrated Testing, Analysis And Verification System

In this section it shall be shown that the strengths and weaknesses of the four techniques just discussed are fortuitously complementary. It shall be seen that it is reasonable to consider the construction of a single comprehensive system capable of verifying a program or efficiently detecting the existence of errors and anomalies, and then employing increasingly powerful tools for pinpointing and examining these errors and anomalies. This increasing exploration will be shown to exploit and encourage increasing amounts of human interaction.

Recently, each of the four techniques has received considerable attention and investigation. Stand-alone systems, implementing each have been constructed, and experience has been gained in using each. Partly as a result of this experience, there is a growing concensus that no single technique adequately meets all program testing verification and analysis needs, but that each contributes some valuable capabilities. It thus becomes clear then that the four techniques should not be viewed as competing approaches, but rather that each offers useful but different capabilities. Attention then naturally turns to the examination of how the various capabilities can be merged into a useful total methodology and system.

Such a methodology is described below. Problems relating to the implementation of the methodology as an Integrated Testing, Analysis and Verification System (ITAVS) are discussed later. The methodology makes provision for the progressive detection and exploration of errors as well as provision for selective verification of different aspects of program behavior.

Both types of activities are begun with static analysis of the source program using a pathwise anomaly detecting analyzer. In the next phase of the methodology, symbolic execution is used to further the results of static analysis. The symbolic execution focuses on anomaly bearing paths detected by the static analyzer to further the error detection and verification power of the methodology. The methodology next calls for the application of either dynamic testing or formal verification. Dynamic testing is used to obtain the most precise but restricted examination of the nature and sources of errors and anomalies whose existence has been determined during the first two

phases. Symbolic execution is used to generate test data for the dynamic test of individual cases. Formal verification is used to obtain the most definitive demonstration of the absence of errors. Extreme rigor and thoroughness can be applied at high cost in showing the absence of errors.

It is anticipated that an ITAVS static analyzer will begin by examining source programs for a wide variety of error and anomaly conditions. Some of these conditions may not be associated with any particular path, but rather with a single node or edge of the program flow graph. In such cases, path generation capabilities such as described in [24] will be used to generate the associated paths. Most of the more interesting conditions, however, are associated with specific paths. Existing systems, such as DAVE [18], demonstrate a number of such interesting conditions and the paths along which they occur can be generated. A formal framework for describing this type of static analysis is presented in Reference 25 and offers hope that other significant families of errors and anomalies and associated paths can be scanned for in similar ways. For example, static analysis can be used to scan for references to unopened files and for division by divisors not checked for zero. The static analyzer will either certify that such conditions do not exist within the program, or will identify paths on which they do occur.

The ITAVS symbolic executor can next accept paths produced by the static analyzer and probe the nature of errors or anomalies more carefully. In the course of symbolically executing a path, the actual executability of the path will be studied. If the path is found to be unexecutable, then the phenomenon occurring along the path can no longer be considered to be of interest, and the path will be removed from further consideration. Other paths to the phenomenon may have to be generated and considered. If the path is found to be executable, then the results of the symbolic execution should shed additional light on the nature and origin of the phenomenon by exposing the derivations of all program variables associated with it.

The symbolic executor can also be used to provide additional assurances about the absence of errors. The symbolic executor will search for such conditions as possible divisions by zero and array bounds violations, which appear difficult to detect infallibly by using static analysis techniques. A path generation subsystem (see for example [26 and 27]) can be used to augment the path set produced by the static analyzer in order to bring to bear the expanded error and anomaly detection power of symbolic execution upon a more comprehensive set of program paths.

For each path symbolically executed and found to be actually executable, a set of test data will be generated, which will be sufficient to effect the execution of the input path. The ITAVS dynamic testing phase will be available to carry out a carefully monitored execution of any path desired by the human tester. It is expected that the information obtained from the static analysis and symbolic execution will suffice to elucidate many errors and anomalies sufficiently so that dynamic testing will not be required to furnish further error exploration capabilities. In these cases, the human tester will make no use of the test input data generated. In other cases, an actual dynamic test may seem useful. One of the primary advantages of performing a dynamic test is that the environment in which the subject program is run is quite close to the actual execution environment. Thus, possible execution errors cannot be hidden by inconsistencies between the assumptions of the symbolic executor and the actualities of the compiler, operating system, and hardware which comprise the execution environment. In short, it seems dangerous to consider a testing regimen to be satisfactory until and unless the subject program has actually been executed in its destination environment. For this reason, many paths to which no errors or anomalies have been associated will be executed by the dynamic testing system. These paths will be drawn from the set of paths generated by the path generation subsystem as input to the symbolic execution system.

The human tester will assume a role of great importance during this phase, deciding which input data sets are to be given to the dynamic testing system, and using the facilities of the system to expeditiously determine the source of observed errors or anomalies. The human will also be responsible for constructing assertion sets to be dynamically verified. These assertions will be similar to those used in a more formal verification of correctness as described below. It is less costly, however, to verify the assertions dynamically for a limited set of test executions than to verify them mathematically. Such a test regimen, moreover, is useful in demonstrating that the assumptions of the mathematical model used as the basis for formal verification correspond to the actual run-time environment. It should be reemphasized that such dynamic verification of individual test cases cannot definitively establish the propriety of all modeling assumptions or the total correctness of the program. Rather, the purpose is to identify gross modeling errors and provide a highly cost effective means to partially verify adherence to intent.

Finally, ITAVS will incorporate a formal verification system to facilitate the formulation of mathematical, comprehensive demonstrations of the absence of any errors. As already noted, this technique is capable of giving the most rigorous and thorough assurances of the absence of error at the highest cost. It is the logical culmination of the process of verifying adherence of a program to stated individual expressions of intent. This process relies upon the ability to perform symbolic execution, and compare the results to assertions. Hence, the methodological approach suggested here is that the symbolic execution phase be used to obtain fragmentary assurances of the absence of error by determining agreement with small assertion sets. More comprehensive verification is obtainable by the human user by verification of

more complete assertion sets. In the extreme cases, this becomes de facto complete formal verification. This process is facilitated and formalized by the formal verification capability of ITAVS.

A schematic diagram of the proposed ITAVS is shown in Figure 1.

It can be seen from the preceding paragraphs that the ITAVS strategy organizes the four techniques into a progression of capabilities which is natural in a number of important ways. The ITAVS begins with a broad scanning procedure and progress to deeper and deeper probing of errors and anomaly phenomena. ITAVS initially requires no human interaction or input. It progresses to involve more significant human interaction as human insight becomes more useful in tracing errors to their sources and constructing mathematical demonstrations of correctness. ITAVS provides the possibility of some primitive verification without human intervention, and then allows error detection based upon the negative results of the verification scan. The flow of data through ITAVS is also most fortuitous. The first phase static analysis requires no input. It produces as output, however, paths through the program which are deemed to be significant in error and anomaly exploration. The second phase of ITAVS, symbolic execution, requires a set of paths as input. This, however, is precisely the nature of the output of the first phase. Finally, the dynamic testing phase requires actual program input data. It has been observed, however, that symbolic execution systems can be used to produce data sets which are sufficient to force the execution of their input paths. Hence, the second phase can be used to provide the input required by the dynamic testing phase.

There also appears to be a natural progression of costs for the processing of the four phases of ITAVS, although exact cost determinations are difficult to make due to the scarcity of production static analyzers and symbolic execution systems. Static analysis appears to be the least expensive, most straightforward operation. It involves a single scan over the program flow graph, using information encoded as bit vectors and algorithms which execute in order n log n time, where n is the number of flow graph vertices. The symbolic execution phase appears far more costly. It entails performing considerable symbol manipulation and constraint solving for each of the presumably many paths generated by both the static analyzer and path generator. Dynamic testing is likewise relatively costly and, perhaps, more involved than the other two. It entails performing a number of closely monitored executions of the program. Each execution may take as much as several times as long as an unmonitored execution [28] and will generate a large data base of diagnostic information which may have to be probed extensively by human interaction. Formal verification appears to be the most costly of the four techniques. It requires extensive amounts of mathematical labor. It is thus reasonably placed as a high-cost option at the end of the methodology.

It is also interesting to observe that the use of assertions provides a unifying influence in integrating the four techniques. All techniques except static analysis use explicit assertions to demonstrate either the presence or absence of errors. Static analysis uses implicit assumptions of proper behavior as embodied in language semantics, but could also benefit from explicit assertions. Seen in this light, the four techniques basically differ in the manner and extent to which they perform assertion verification. Thus, it seems reasonable to require that a program and initial set of assertions be submitted to ITAVS. The adherence of program to assertions would be examined at every stage of ITAVS. The stages would test the adherence in different ways, progressively establishing firmer assurances of adherence or focusing more sharply on deviations.

## IV. Problems And Future Research

The foregoing section has presented a rather sanguine view of the capabilities of an integrated testing system combining the best features of static analysis, symbolic execution, dynamic testing, and formal verification. Although software systems implementing each of these techniques have been produced, the task of constructing a usable ITAVS is still far more formidable than simply building software interfaces between existing systems. Significant research must be completed before a useful ITAVS can be built. It is the purpose of this section to discuss a number of these research areas and in some cases to propose avenues of approach to their solution.

The overriding goal of this body of research should be to determine how verification and error detection capabilities can best be distributed among the four techniques. The capabilities of each technique have been indicated briefly in previous sections, and investigators in each area have shown that these basic capabilities can often be expanded. The focus of this research, however, should be not on what can be done by each individual technique, but rather on what should be done by each in the context of the overall system. In order to determine this, it seems important not only to continue the exploration of the basic capabilities to the desired overall capabilities of an ITAVS and to explore the significant problems involved in interfacing the techniques to each other.

Many of the most important and challenging research questions surround the integration of symbolic execution into an ITAVS. The research already done into the strengths and weaknesses of symbolic execution [29] should form a basis for some of this work. For example, it is recognized that problems often arise in attempting to symbolically execute a program path which references arrays. It is known, moreover, that in some cases the symbolic execution cannot safely proceed beyond such references, while there is a range of other cases for which the symbolic execution is possible, but at the cost of increasingly difficult analysis. It is important to determine how severely the validity

of symbolic execution as a testing tool is restricted by the problems posed by array references.

Another challenging and important research matter is the study of the extent to which the symbolic execution system can be relied upon to assist in determining the exact test paths which should be executed to obtain the optimal diagnostic value. Static analysis systems can be used to provide test paths, but the paths which they supply have significant failings. First, static analyzers currently make no attempt at determining whether the paths which they generate are executable or not. Symbolic execution systems can often make this determination, but it would be better if at least some non-executable paths were screened out by the static analyzer. Some advances in this direction have been made, and will be discussed later. Another approach to this problem would be to have the static analyzer supply not an entire path, but rather a specification of a path. This specification would include sufficient information (e.g. a sequence of nodes and/or edges of the flow graph) to describe a family of paths along which anomalous or erroneous behavior would necessarily take place, but would leave the exact specification of the path to the symbolic execution system. The symbolic execution system, because it has access to information about the relations between values of program variables, is in a good position to construct executable program paths within the constraints imposed by the static analyzer, even as the symbolic execution proceeds. Clearly, the exact mechanisms for conveying the path constraints must be studied, as must the procedures to be used by the symbolic execution system in constructing an actual path.

A second significant failing of the paths supplied by static analysis systems is that they contain no information about how many tines loops should be iterated. It is easy to see, however, that certain errors occur only after a certain number of loop iterations. Static analysis systems do not have sufficient information about the values of variables to determine the number of loop iterations necessary to force errors. Symbolic execution systems seem to contain the necessary information. The exact mechanism needed to use that information to synthesize loop counts must be studied. There is hope, however, that some of the current work in the area of using symbolic execution to synthesize loop invariants [30] will be useful here. Hence, techniques for synthesizing loop invariants should be studied and mechanisms for using them to compute loop counts which force errors and anomalies should be examined.

There are also many crucial research questions surrounding the integration of a static analyzer into the ITAVS. As already noted, the static analyzer is expected to supply to the rest of the ITAVS a set of program test paths along which anomalous or erroneous situations will arise. It has also been observed that the test paths produced by current static analysis systems [16, 17, and 18] are not satisfactory in the context of an ITAVS. The paths are inadequate because static analyzers do not currently scan for a rich enough variety of error and anomaly conditions, and because these systems make no effort at suppressing unexecutable paths. It is crucial that these problems be attacked, because the static analyzer is the front end for the ITAVS, and it would be extremely difficult for subsequent systems to compensate for its inadequacies in scanning for phenomena to be investigated more closely, or in screening out inherently uninteresting phenomena.

Thus, it is encouraging to observe that some additional error and anomaly conditions can be detected by current static data flow analysis techniques. For example, consider the problem of detecting the possibility of division by zero. In order to protect a division operation from the possibility of division by zero, it is often advisable to precede the division with a test of the divisor for equality to zero. An unsafe division is one not preceded, along some path, by such an event sequence. It seems clear that a static data flow analysis algorithm could be used to detect both safe and unsafe divisions by scanning a flow graph similar to that used in DAVE [18].

Other error and anomaly conditions should likewise be detectable to some extent by static analysis similar to that used in DAVE. Array subscript bounds violations, for example, should be detectable by such a procedure. Another possibility is the detection of possible change in the data type by which a given variable is referenced. The best application of this type of checking would probably be to some language such as SNOBOL4 [31]. In this language, there is a wide variety of data types and, due in part to the highly interpretive nature of the language, many data type conversions are done automatically by the interpreter. Here the detection of improper changes in the data type of a given variable is far harder for a static analyzer. Such checking could probably be done, however, with static data flow analysis algorithms like those just described.

The preceding makes it clear that it should be both important and fruitful to pursue research into the range and diversity of error and anomaly conditions whose detection can be effected by data flow analysis.

With the realization, moreover, that so much valuable static analysis can be carried out by adaptations of a few basic data flow analysis algorithms, comes an appreciation of the pivotal importance of these algorithms. Hence, the further study of these algorithms must be recognized as a closely allied research area of importance.

As alluded to earlier, another important research area is the study of how the static analysis system can recognize unexecutable paths and suppress their transmission to the symbolic execution system. It is theoretically impossible to always detect whether or not an arbitrary path through an arbitrary program is executable. It has already been observed, moreover, that symbolic execution systems have other difficulties

in making this determination. Hence, it appears that here, as in the detection of conditions such as unsafe division, what is needed is a mechanism within the static analyzer for filtering out as many unexecutable paths as possible, rather than attempting a definitive resolution of the executability question. Here too, there is considerable hope of constructing such a filter, and the mechanism, fortuitously, appears to be the same type of data flow analysis algorithms as those used in error and anomaly detection. It was noted earlier that a path is unexecutable by virtue of the fact that the constraints upon its input data posed by its transfer conditions are unsatisfiable by any input data. Hence, the executability of a path can be studied by examining its transfer conditions. There are some simple cases in which this examination can be done quite readily. For example, suppose a program contains two lexically identical IF statements and that along all paths from one to the other, the values of all statement variables are left unchanged. Then, clearly, the outcome of the second IF test must always be the same as the outcome of the first IF test for any execution. A pair of transfers representing opposite outcomes from two such tests is called an impossible pair (see [32]). Clearly any program path containing both transfers of an impossible pair must be unexecutable. Thus knowledge of the impossible pairs of a program can be used to filter out unexecutable paths. It appears that data flow analysis algorithms such as those proposed above for inclusion in the static analysis system of an ITAVS can also be used to detect impossible pairs within a program. This would enable the static analyzer to perform some filtering out of unexecutable paths.

The mechanism by which data flow analysis can be used to generate impossible pairs is discussed in Reference 33. That approach should result in the production of sufficient information to inhibit the generation of some unexecutable paths by the static analyzer. There must be additional attention devoted, however, to the question of how this information will be utilized in inhibiting these paths. It has already been observed that impossible pairs can be used as filters to reject paths already generated. This, however, is a less satisfactory approach than utilizing the same information to somehow avoid the generation of the unexecutable paths as the path generation process is proceeding. Because the path generation process is necessarily an outgrowth of the data flow analysis which is performed as part of the error and anomaly detection described earlier, it is hoped that the error/anomaly scan might be guided away from unexecutable paths by a simultaneously ongoing data flow scan for impossible paths. This path suppression idea should be explored. Another possible use of the impossible pairs information is to supply it to the symbolic execution system along with an incomplete path specification, as described earlier, and allow the symbolic execution system to use this information to in some way guide the creation of complete executable paths from the incomplete specifications.

All of the above questions are actually different facets of the more overriding question of how much effort should be devoted by the static analyzer to the suppression of unexecutable paths, and how much should remain within the symbolic execution system. No matter what decision is reached, however, it must be expected that the static analyzer will sometimes pass to the symbolic execution system paths which are subsequently found to be unexecutable. This poses an important problem, which must also be considered carefully. Because the path was generated by the static analyzer, it must be assumed that its execution would have caused an error or anomaly. In finding that the path is unexecutable, however, it is not proven that the error or anomaly cannot occur, it is only proven that the error or anomaly cannot occur along the given path. Hence, there remains the question of whether an executable path might not be constructible for which the same error or anomaly would occur. It appears that the search for such a new executable path might be guided to some extent by the information gathered by the symbolic execution system in the process of detecting the unexecutability of the original path. The unexecutability finding arises from the discovery that a set of constraints is inconsistent. More specifically, it must be expected that there is a relatively small subset of the constraints (probably only two) which is inconsistent. Hence, some information about the source of the unexecutability can be made available and used to guide subsequent path generation efforts. The exact mechanism for profitably utilizing this information should be the object of further research, and, depending upon the outcome, the generation of subsequent paths might be carried out within the symbolic execution system itself, or within the static analyzer.

Finally, there are a number of problems which should be addressed in integrating the dynamic testing system into an ITAVS. First and most obviously, the test data produced by the symbolic execution system must be formatted in such a way that it can be read correctly as input by the instrumented program in order to carry out the dynamic test. There should also be a communications protocol between the two systems so that duplication of testing is avoided. For example, user supplied assertions which have been verified during symbolic execution should not be retested dynamically, while those which could not be verified by symbolic execution (due, presumably, to intractible array references and/or systems of constraints), should be tested dynamically. Another research area which should be pursued is the investigation of data base accessing systems as tools for building and examining the execution history generated by dynamic testing. Much of the cost of performing dynamic testing might be avoided if the evolving technology for handling large data bases were applied directly to the management of the execution history.

## V. Conclusions

There are numerous obstacles to the creation of a practical ITAVS. A number of the most pressing of these obstacles have been presented here. There appears to be good reason, however, to expect that the solutions to many of them are within reach today. Clearly a great deal of

research is indicated. The primary goal of this research should at least for the near term, be the determination of which testing functions should be performed by which testing subsystems, and how these subsystems can be merged into an optimally comprehensive yet efficient system.

The outlines of some of the longer range outcomes of this line of research can be observed already. It appears, for example, that this research will show that many of the testing operations currently performed by dynamic testing systems alone, can be performed more effectively by some combination with static analysis, symbolic execution and formal verification. This would lessen the reliance of testing upon chance and human interaction. It also appears that this research will show that the activities of program testing and formal verification are more closely related than previously generally thought. Some of the static analysis techniques proposed here can reasonably be thought of as techniques for producing proofs of the correctness of certain restricted aspects of a given program. Moreover, certain proposed applications of symbolic execution are tantamount to assertion verification over a limited range. It is expected that this research may provide some insight into some ways in which testing and proving activities can be utilized as complementary activities. The proposed research should confirm these and other important conjectures.

## VI. Acknowledgments

The author would like to thank Lloyd D. Fosdick for the many valuable and stimulating conversations which helped shape the ideas presented here, as well as for his perceptive comments on early versions of this paper. The ideas presented here were also shaped by stimulating conversations with Lori Clarke, Bill Howden, Jim King, Don Reifer, Dick Fairley, Leon Stucki, Bob Hoffman, and many others.

## VII. References

[1] F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal (11) pp. 56-73 (1972).

[2] D. S. Alberts, "The Economics of Software Quality Assurance," AFIPS Conference Proceedings (45) pp. 433-442 (1976 National Computer Conference).

[3] E. W. Dijkstra, "Notes on Structured Programming", in Structured Programming by O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Academic Press, London and New York, 1972.

[4] N. Wirth, "Program Development by Stepwise Refinement" CACM 14 pp. 221-227 (April 1971).

[5] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems Into Modules" CACM 15 pp. 1053-1058 (December 1972).

[6] N. Wirth, "An Assessment of the Programming Language PASCAL" IEEE Transactions on Software Engineering SE-1 pp. 192-198 (June 1975).

[7] J. D. Gannon and J. J. Horning, "Language Design for Program Reliability", IEEE Transactions on Software Engineering SE-1 pp. 179-191 (June 1975).

[8] R. M. Balzer, "EXDAMS: Extendable Debugging and Monitoring System", AFIPS 1969 SJCC 34 AFIPS Press, Montvale, New Jersey pp. 567-580.

[9] R. E. Fairley, "An Experimental Program Testing Facility", Proceedings of the First National Conference on Software Engineering, IEEE Cat #75CH0992-8C, pp. 47-52.

[10] L. G. Stucki and G. L. Foshee, "New Assertion Concepts for Self Metric Software Validation", Proceedings 1975 International Conference on Reliable Software, IEEE Cat. #75CH094C-7CSR, pp. 59-71.

[11] R. Grishman, "The Debugging System AIDS", AFIPS 1970 SJCC 36 AFIPS Press, Montvale, N. J. pp. 59-64.

[12] L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering SE-2 pp. 215-222 (September 1976).

[13] W. E. Howden, "Experiments With A Symbolic Evaluation System", AFIPS 1976 NCC 45 AFIPS Press, Montvale, N. J. pp. 899-908.

[14] J. C. King, "Symbolic Execution and Program Testing", CACM 19 pp. 385-394 (July 1976).

[15] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution", Proceedings 1975 International Conference on Reliable Software, IEEE Cat. #75CH094D-7CSR, pp. 234-245.

[16] C. V. Ramamoorthy and S. B. F. Ho, "Testing Large Software With Automated Software Evaluation Systems", IEEE Transactions on Software Engineering SE-1 pp. 46-58.

[17] E. F. Miller, Jr., "RXVP, Fortran Automated Verification System", Program Validation Project, General Research Corporation, Santa Barbara, California (October 1974).

[18] L. J. Osterweil and L. D. Fosdick, "DAVE--A Validation, Error Detection, and Documentation System for Fortran Programs", Software Practice and Experience 6 pp. 473-486.

[19] E. F. Allen and J. Cocke, "A Program Data Flow Analysis Procedure", CACM 19 pp. 137-147 (March 1976).

[20] K. W. Kennedy, "Node Listings Applied to Data Flow Analysis", Proceedings of 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, California pp. 10-21 (January 1975).

[21] M. S. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems", SIAM J. Computing 4 pp. 519-532 (December 1975).

[22] J. D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions", Acta Informatica 2 pp. 1910213 (December 1973).

[23] A. L. Ambler, D. I. Good, W. F. Burger, "Report on the Language GYPSY," Certifiable Minicomputer Project, Institute for Computing Sciences and Computer Applications, University of Texas at Austin, #ICSCA-CMP-1, August 1976.

[24] L. J. Osterweil, "Depth First Search Techniques and Efficient Methods for Creating Test Paths", University of Colorado Department of Computer Science Technical Report # CU-CS-077-75 (August 1975).

[25] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability", ACM Computing Surveys 8 pp. 305-330.

[26] J. C. Huang, "An Approach to Program Testing", ACM Computing Surveys 7 pp. 113-128 (September 1975).

[27] R. H. Hoffman, "NASA/Johnson Space Center Approach to Automated Test Data Generation", Proceedings of the Computer Science and Statistics Eighth Annual Symposium on the Interface, Los Angeles, California, pp. 336-341 (February 1975).

[28] R. E. Fairley, "Dynamic Testing of Simulation Software", Proceedings of 1976 Summer Conference Simulation Conference, Washington, D. C. pp. 708-710.

[29] L. Clarke, "Test Data Generation and Symbolic Execution of Programs as an Aid to Program Validation", University of Colorado Department of Computer Science Ph.D. Thesis (August 1976).

[30] B. Elspas, "The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness" Interim Report, SRI Project 2686, Stanford Research Institute, Menlo Park, California (July 1974).

[31] R. E. Griswold, J. F. Poage and I. P. Polonsky, The SNOBOL 4 Programming Language, Prentice-Hall, Englewood Cliffs, N. J. Second Edition 1971.

[32] K. A. Krause, R. W. Smith and M. A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis", 1973 IEEE Symposium on Computer Software Reliability, IEEE Cat. #73C40741-9CSR New York, pp. 18-22 (June 1973).

[33] L. J. Osterweil, "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis," Proceedings of COMPSAC 77, IEEE Computer Society, November 1977, Chicago, Illinois.

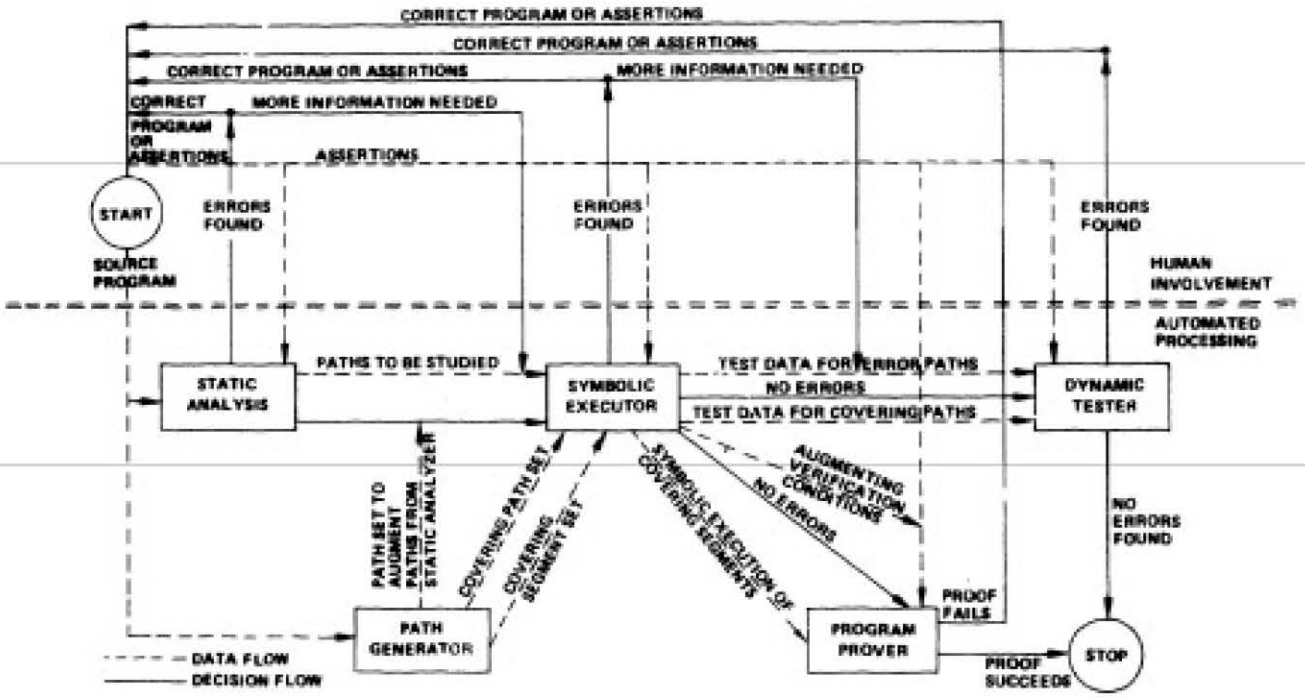



FIGURE 1. A DIAGRAM OF THE PROPOSED ITAVS