A Proposal for an Integrated Testing System for Computer Programs

Leon J. Osterweil

CU-CS-093-76 August 1976

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A PROPOSAL FOR AN INTEGRATED TESTING SYSTEM FOR COMPUTER PROGRAMS

by

Leon J. Osterweil
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-093-76

August, 1976
Revised October, 1976

This work supported by NSF Grant #DCR75-90072

Abstract

The idea of combining three well known program testing techniques--dynamic testing, symbolic execution and static analysis--into a single testing system is advanced and explored here. The characteristics of each of the three 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 a number of difficulties to be expected in implementing such a system are discussed. Approaches to resolving many of these difficulties are also presented.

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: proof of correctness of programs [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 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. In this paper we shall refer to the former activity as *error detection* and the latter as *validation*. Among the diverse activities in the areas of error detection and validation, we shall focus on three of the major approaches -- namely dynamic testing, symbolic execution, and static analysis. In this paper the different patterns of strengths, weaknesses and applications of these approaches are shown. It is, moreover, 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 system capable of performing a diverse and useful variety of error detection and validation functions. This paper also explores some of the problems involved in integrating the methodologies with one another, indicating areas where a great deal of valuable and challenging research remains to be done.

II. THREE PROGRAM TESTING METHODOLOGIES

In this section, the different characteristics of dynamic testing, symbolic execution, 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 (see for example [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 previous paragraphs should make it clear that dynamic testing systems have strong error detection capabilities due not simply to their powerful facilities for recognizing errors during the execution of a program, but also for 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 of a program does not guarantee their absence in the program itself. It is thus seen that dynamic testing systems have no inherent validation capabilities.

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 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 required in order to effectively use the capabilities of the system to detect and explore execution errors. Hence it is seen that the power to obtain detailed insight comes in part from significant involvement of the human program tester.

Thus, summarizing, it has been shown that dynamic testing systems provide strong error detection capabilities, but have no inherent validation capabilities. 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. 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 a procedure is being tested, or read-in values); all other values 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 validation 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 the program's input values. 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 validation. 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 validation. Error detection for program computations can be achieved 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 seem to have some validation-like 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 validation*. Clearly pathwise validation falls short of complete validation, 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 validation 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 on 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 validation capabilities, which fall short of the power of full validation. 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 perform their analysis along specific paths. Therefore, since they are expensive systems to run, it is desirable to restrict their application to carefully selected paths in a program, paths which are suspected to contain errors or are otherwise provocative. This (often) requires 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 (see for example [19], [20], [21], and [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 preceeding paragraphs that static analysis systems offer a limited amount of error detection, but are capable of performing certain validation 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 validation 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. In an important sense it can be seen 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.

III. AN INTEGRATED TESTING SYSTEM

In this section it shall be shown that the strengths and weaknesses of the three methodologies just discussed are fortuitously complementary. It shall be seen that it is reasonable to consider the construction of a single comprehensive system capable of validating a program (in some significant sense), efficiently detecting the existence of errors and anomalies, and then employing increasingly powerful tools for pinpointing and examining these errors and anomalies. This progressive exploration, moreover, would exploit and encourage increasing amounts of human interaction.

In the recent past, each of the three testing methodologies has received considerable attention and investigation. Stand-alone systems, implementing each methodology 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 methodology adequately meets all program testing needs, but that each contributes some valuable capabilities. It thus becomes clear that the three methodologies should not be viewed as competing approaches, but rather that each is a contributor to the body of needed testing tools. Attention then naturally turns to the examination of how the various contributions can be melded into a useful total system.

The strategy for such an Integrated Testing System (ITS) should be to begin with static analysis of the source program, thereby obtaining validation results and broad error and anomaly detection capabilities; proceed to a symbolic execution phase, thereby obtaining more detailed and powerful error and anomaly detection capabilities focused largely upon program paths already known to contain interesting phenomena; and finally carry out dynamic testing of the program in order to obtain the most precise examination of the nature and sources of errors and anomalies whose existence has previously been determined.

It is anticipated that an Integrated Testing System (ITS) 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 [23] 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 that 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 [24] and offers hope that other significant families of errors and anomalies and associated paths can be scanned for in similar ways.

The ITS symbolic execution system will then accept the paths produced by the static analyzer and probe the nature of the error or anomaly more carefully. In the course of symbolically executing a path, the executability of the path will be determined. 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. In addition, the symbolic execution 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 [25] and [26]) may 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 executable, a set of test data will be generated, which will be sufficient to effect the execution of the input path.

The ITS 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 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 actualitites 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.

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

It can be seen from the preceeding paragraphs that the ITS strategy organizes the three methodologies into a progression of capabilities which is natural in a number of important ways. The ITS begins with a broad scanning procedure and progress to deeper and deeper probing of errors and anomaly phenomena. ITS initially requires no human interaction or input, and progresses to involve more significant human interaction as human insight becomes more useful in tracing errors to their sources. ITS provides the possibility of validation without human intervention, and then allows error detection based upon the negative results of the validation scan. The flow of data through ITS 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 ITS, symbolic execution, requires a set of paths as input. This, however, is precisely the nature of the output of the first phase. Finally, the third phase, dynamic testing, requires actual program input data. It has been observed, however, that symbolic execution systems can be used to produce data sets which are sufficient

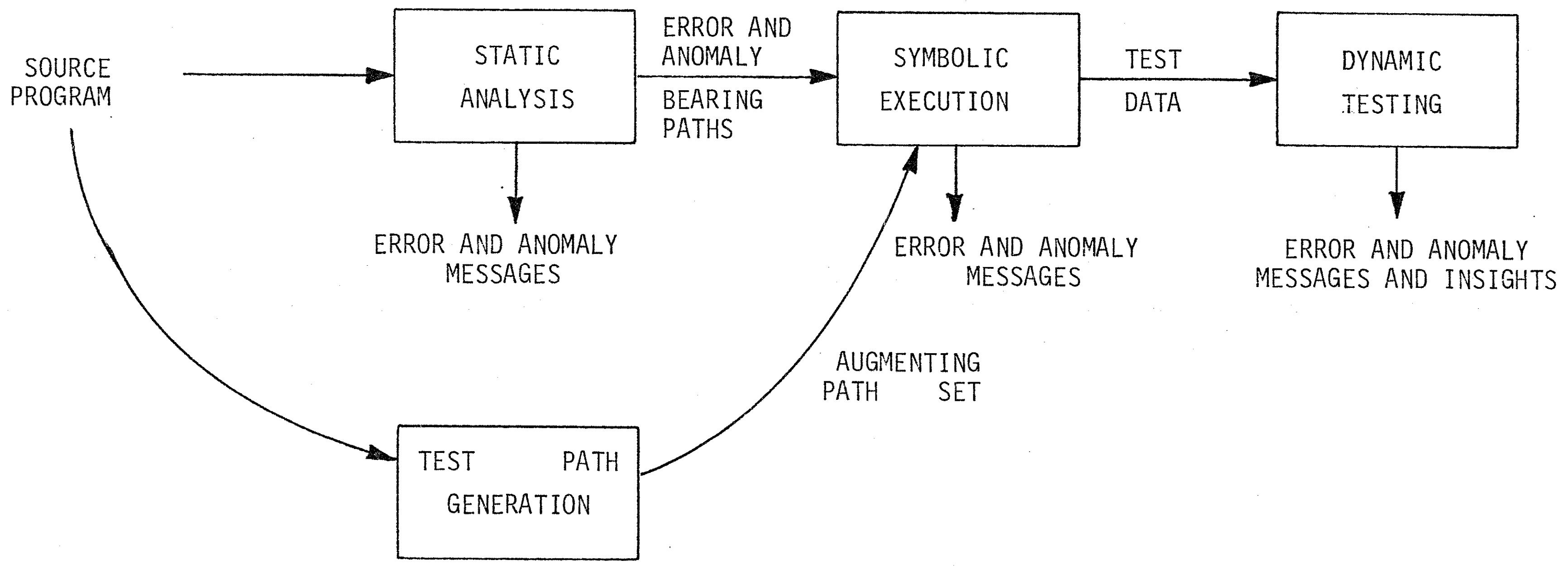
SOURCE PROGRAM
STATIC ANALYSIS
ERROR AND ANOMALY BEARING PATHS
SYMBOLIC EXECUTION
TEST DATA
DYNAMIC TESTING
ERROR AND ANOMALY MESSAGES
ERROR AND ANOMALY MESSAGES
ERROR AND ANOMALY MESSAGES AND INSIGHTS
AUGMENTING PATH SET
TEST PATH GENERATION

Figure 1: A schematic diagram of the proposed ITS.

to force the execution of their input paths. Hence the second phase can be used to provide the input required by the third phase.

There also appears to be a natural progression of costs for the processing of the three phases of the ITS, 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 flowgraph 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, the most involved process of the three. 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 [27] and will generate a large data base of diagnostic information which may have to be probed extensively by human interaction.

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, and dynamic testing. Although software systems implementing each of these three methodologies have been produced, the task of constructing a usable ITS is still far more formidable than simply building software interfaces between existing systems. Significant research must be completed before a useful ITS 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 which validation and error detection capabilities should be performed by which testing methodologies. The capabilites of each methodology have been indicated briefly in previous sections, and investigators in each area have shown that these basic capabilites can often be expanded. The focus of this research, however, should be not on what can be done by each individual methodology, 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 of each methodology, but also to study the relationship of these capabilities to the desired overall capabilities of an ITS, and to explore the significant problems involved in interfacing the methodologies to each other.

Many of the most important and challenging research questions surround the integration of symbolic execution into an ITS. The research already done into the strengths and weaknesses of symbolic execution (e.g. see [28]) 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 analysis of increasing difficulty. 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. An empirical study of actual programs would be able to determine a profile indicating a distribution of expected array reference patterns. This profile would be helpful in formulating a strategy for producing a set of heuristic routines for enabling the symbolic execution of most program paths containing array references. The empirical study would indicate which existing heuristics should be incorporated into the symbolic execution system of a practical ITS, and might also indicate cases for which new heuristics are needed. In the same vein, it has been observed [28] that even if the symbolic execution of a particular program path is completed successfully, the constraints generated may nevertheless be so complex as to prohibit solution, thereby eliminating the possibility of generating test data. It is observed that this is due to a fundamental theoretical impossibility, rather than an engineering difficulty. Hence, from the point of view of an ITS this implies that there will be program test paths for which the symbolic execution system will be unable to generate input to the dynamic testing system.

In Clarke's system, test data generation is only attempted in case the constraints imposed by the path through the program upon the input values are found to all be linear. In this case linear programming techniques are used to solve the system of constraints. Here too, an empirical study would be quite useful in determining how often non-linear constraints can be expected to occur. In any case, it seems necessary to consider methods for dealing with non-linear constraints. Heuristic approaches to the solution of systems of non-linear constraints have been studied (see [29] for example), and should be investigated further from the point of view of practicality in an ITS. In addition, recent encouraging results with interactive theorem provers [1] seem to indicate that human intervention is often extremely helpful to automatic programming systems which manipulate formal mathematical systems. Hence, some form of human intervention in the process of solving non-linear constraints should be considered in designing an ITS symbolic execution subsystem.

Another area that will be studied further is procedure invocation. Current symbolic execution systems either ignore procedure calls or analyze a procedure path each time the procedure is invoked. Neither method is satisfactory. The first method does not detect the effect the called procedure has on the remaining path. The second method is inefficient, often requiring that a path be analyzed repeatedly. Techniques are needed therefore to encapsulate a procedure's input and output behavior so that its consistency with the existing path may be verified and its effect on the continuing symbolic execution conveyed. Methods of encapsulating a set of paths for each procedure, selecting an appropriate path when the procedure is invoked, and incorporating the encapsulated information into the calling procedure must be investigated.

There must also be further investigation of the classes of errors which are detectable by means of symbolic execution. Clarke's system is able to detect the possibility (or impossibility) of the occurrence along a given path through a program, of such errors as division by zero, and out of bounds values for such variables as subscripts, DO statement parameters, and computed GO TO indices. It has been shown that the overall error detection and validation capabilites of the ITS have been significantly enhanced in this way. It is important to determine whether there are other classes of errors which can be detected and validated for in this way. In particular, it seems that the technique employed by Clarke, namely the creation and insertion of a new constraint embodying the error condition, could be used to detect any error which could occur as the result of an improper value for some variable or set of variables. There are undoubtedly other language constructs in FORTRAN and in other languages which require certain variables to assume restricted sets of values. A carefully organized, thorough study should be undertaken to determine the nature of all such errors and the mechanisms by which symbolic execution can be used to detect them and validate for their absence.

Perhaps more important, however, it seems that the process of comparing programmer-defined assertions to the actual workings of the program could also be performed as an application of the constraint generation and solution process during symbolic execution. Assertion verification has previously been described as a capability of some dynamic testing systems ([19], [10]) and King's EFFIGY symbolic execution system [14]. The advantage of attempting assertion verification during symbolic execution would be that the symbolic execution system could carry out pathwise validation for the impossibility of violating such assertions, while the dynamic tester can only perform error detection. The idea of applying symbolic execution to assertion verification is not totally new here. In EFFIGY, symbolic execution is used to simplify expressions and thereby facilitate the theorem proving processes needed to verify assertions contributing to proof of the program's correctness. A similar approach is taken by Deutsch in the PIVOT system [36], which sometimes attempts to verify assertions along specific individual paths. Neither system, however, seems to exploit the possibility of using numerical constraint solving techniques to verify assertions. It appears that many significant assertions can be stated as relations between the values of sets of program variables. Formulas embodying these assertions could thus be supplied by the user and could then be compared to formulas generated by the symbolic execution system as the actual relations between the given program variables. Determination of the equivalence of the given and actual relations could be attempted both by direct comparison of formulas and through the solution of equivalence constraints. It is recognized that this equivalence determination will not be possible in some cases, but it is expected that it will be feasible for significant classes of assertions. Here too, a study should be made to classify the assertion checking capabilities which can be readily performed by a symbolic execution system to determine how well these capabilities meet anticipated needs.

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 times loops should be iterated. It is easy to see, however, that certain errors occur only after a certain number of loop iterations. Consider the sequence of code in Figure 2, for example. Here it is easy to see that an array subscript bounds violation will occur only after the loop has been iterated more than 100 times. Static analysis systems may be able to detect the possibility of such subscript errors in some cases. More will be said about this later. It is certain, however, that even should static analysis systems be able to detect

```
       SUBROUTINE A (ARRAY, N)
       DIMENSION ARRAY(100)
       DO 10 I=1, N
    10 ARRAY(I)=0
       RETURN
       END
```

Figure 2: A subroutine for which an error will occur only after the loop is iterated more than 100 times.

the possibility of such errors, they would not have sufficient information about the values of variables to determine the number of loop iterations necessary to force the error. 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 (e.g. [30]) will be useful here. In the subroutine shown in Figure 2, for example, it is easy to synthesize the information that the subscript in statement 10 increases by 1 for every iteration of the loop. Coupled with the information that the initial subscript value is 1, this makes it clear that on the 101st iteration of the loop a subscript error will occur. Hence a constraint, constraining the value of the loop termination variable to be 101 or greater, could be created. The symbolic execution system could then determine that this constraint, $N \geq 101$, is consistent with the other constraints and that the error could occur. This appears to be a very valuable type of diagnostic capability. 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 ITS. As already noted, the static analyzer is expected to supply to the rest of the ITS 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 (e.g. [16], [17], and [18]) are not satisfactory in the context of an ITS. 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 crucially important that these problems be attacked, because the static analyzer is the front end for the ITS, 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 pheonomena.

There seems to be considerable cause for optimism that the static analyzer can be used as a scanner for wider classes or errors and anomalies. There are well known algorithms, currently in use in program optimization, which are capable of efficiently scanning all possible paths through a program flow graph for the existence of paths having certain specific patterns of tagged nodes. The algorithms operate only on graphs whose nodes are tagged with only two different kinds of labels. It appears that these algorithms are applicable to the detection of a number of anomalies and errors which are characterized as specific sequences of events along paths through a two-tagged graph. One example of an error of this type is an attempt to write on, or read from, a file which has not previously been opened (such errors are possible in languages such as COBOL). Here the flow graph of the program would be annotated by two different labels -- open and reference. The sequence which would be scanned for would be a reference not preceded by an open. Other such examples can likewise be produced.

It is more interesting, however, to consider the class of errors and anomalies which are naturally described as sequences of tags along paths through graphs having more than two types of tags. The DAVE system [18] provides a good model of what can be accomplished. In DAVE, the nodes of the program flow graph are tagged with three different kinds of labels indicating reference, definition, and undefinition of program variables. In [23] it is shown that this three-tagged graph can be transformed into a two-tagged graph and analyzed by the data flow analysis algorithms referred to above. These algorithms are used to detect patterns of tags along paths in the two-tagged graph whose analogs in the original three-tagged graph correspond to such errors as referencing an undefined variable, and to such anomalies as defining a variable without subsequently referencing it.

Many other worthwhile phenomena can apparently be modelled as sequences of events along paths in such graphs with higher order taggings. 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 advisable to precede the division with a test of the divisor for equality to zero. Restating this, a *protected division* operation is one such that on all paths leading to the division, there is a test of the divisor against zero which precedes the division, and there is no intervening redefinition of any of the variables contained in the expression comprising the divisor. An *unsafe division* is one not preceded, along some path, by such an event sequence. Hence we see that it is possible to recognize a protected division by zero by examining the sequences of the three events -- division by an expression, testing of the expression, and redefinition of expression variables -- along all program paths. It seems likely that a data flow analysis algorithm could be used to detect both unsafe and protected divisions by scanning a two-tagged graph which had been derived from this three-tagged graph by transformations similar to those used in DAVE.

Hence, unsafe divisions, those which lie on at least one path for which the division is inadequately protected from the possibility of a zero divisor, could be detected by a static analyzer. A path to such a division could likewise be generated and passed to the symbolic execution system for more definitive study by the constraint solving process. It is important to acknowledge that some of the division operations passed to the symbolic execution system in this way may not be unsafe at all. This is quite conceivable if the program tests an expression which is equivalent (but not identical) to the divisor against zero before performing the divison operation. In such cases, the symbolic execution system will be able to determine that the division is protected. Hence it is seen that the use of the symbolic execution system to test for possible division by zero is still necessary, and has not been negated by this procedure within the static analyzer. Rather, the static analyzer is acting as a filter, relieving the symbolic execution system from the need to consider certain paths. Hopefully the static analyzer will act as a fine filter, removing most of the paths which contain protected divisions. Experience and experimentation should demonstrate whether this is so, or whether stronger static analysis should be performed in an attempt to improve the path filtering capability.

Other error and anomaly conditions should likewise be detectable to some extent by static analysis provided that the flow graph can be annotated with larger numbers of tags. Array subscript bounds violations should be detectable by a procedure analogous to the one outlined above for detecting unsafe divisions. Another possibility is the detection of possible change in the data type by which a given variable is referenced. In FORTRAN, there is little possibility of referencing a variable with the wrong data type, as most such references are readily detected by the compiler. One case in which this error is possible involves reading in a variable under one type conversion, and then subsequently referencing the variable under another data type. Howden [31] notes that this error is the cause of program failure in some of the sample programs in Kernighan and Plauger [32], lending weight to the suspicion that this is an error that is worth checking for in FORTRAN programs. This checking could probably be done by techniques which are less sophisticated than data flow analysis, however. The best application of this type of checking would probably be to some other language such as SNOBOL4 [33]. 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 data flow analysis algorithms like those just described.

The preceding paragraphs make 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. This will be an important contribution to understanding the proper relations and interactions between the static analysis system and the symbolic execution system in an ITS.

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. In addition, it appears that it would be quite interesting to study algorithms for performing data flow analysis directly on graphs for which the nodes carry more than two types of tags. It seems clear that algorithms for analyzing such graphs can be produced as adaptations of the algorithms for studying two-tagged graphs. It is important to consider, however, whether other algorithms might not be devised which might be more efficient or better suited to the overall constraints of an ITS.

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 has already been observed that it is theoretically impossible to always detect whether an arbitrary path through an arbitrary program is executable or not. Moreover, it has also been observed 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 IF statements which test predicates which are lexically identical, and that along all paths from one predicate to the other the values of all predicate 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 [34]). 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 ITS, 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 can be demonstrated in the following way. Suppose that each edge of a program flow graph is annotated with the predicate which must be satisfied in order to effect the traversal of the edge. Further suppose that each node of the flow graph is annotated with the identities of all variables whose values are reset by computations performed within the node. Finally suppose that all pairs of incompatible predicates have been identified. Clearly the graph has been annotated with two kinds of tags -- the predicate test tag and the variable reset tag. The data flow analysis procedure now must recognize sequences of edges and nodes which start with a predicate edge, and end at an edge labelled with an incompatible predicate, and which pass through no nodes which reset the value of predicate variables. This data flow analysis procedure differs from the others in that it must examine both nodes and edges. Moreover, the labels themselves appear to be somewhat more complicated. Nevertheless, the algorithms for analyzing such graphs appear to be inherently little different from those needed to perform error and anomaly scans, as described earlier. It is expected that other concepts, similar to the impossible pairs concept, will emerge from the study and creation of this family of algorithms. In addition, it seems that these techniques should also enable the identification of certain edges (such as DO-loop fall through edges) as being unexecutable in some cases. It is also expected that the conditions under which impossible pairs can be identified might be extended by annotating the flow graph more completely with additional program phenomena, and by increasing the sophistication of the data flow analysis algorithms used. This may become desirable if experience indicates that the static analyzer still produces too many unexecutable paths.

All of these approaches 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. This approach would almost certainly necessitate the use of heuristics, because it has already been demonstrated that just the generation of any path constrained to obey impossible pairs can be expected to be a lengthy process (this has actually been shown to be an NP-Complete problem [35]).

All of the above questions are actually differing 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 are 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 ITS. First and most obviously, the test data produced by the symbolic execution system must be formatted in such a way that it can be correctly read 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 practically useful ITS. 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 can be performed more effectively by static analysis and symbolic execution, lessening the reliance of testing upon chance and human interaction. It also appears that this research will show that the activities of program testing and program proving 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, Dick Fairley, Leon Stucki, Bob Hoffman, and many others. Finally, the support of the National Science Foundation in funding this work is most gratefully acknowledged.

VII. REFERENCES

[1] D. I. Good, R. L. London, and W. W. Bledsoe, "An Interactive Program Verification System", IEEE Transactions on Software Engineering, SE-1 pp. 59-67 (March 1975).

[2] B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys 4 pp. 97-147 (June 1972).

[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.

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 can be performed more effectively by static analysis and symbolic execution, lessening the reliance of testing upon chance and human interaction. It also appears that this research will show that the activities of program testing and program proving 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, Dick Fairley, Leon Stucki, Bob Hoffman, and many others. Finally, the support of the National Science Foundation in funding this work is most gratefully acknowledged.

VII. REFERENCES

[1] D. I. Good, R. L. London, and W. W. Bledsoe, "An Interactive Program Verification System", IEEE Transactions on Software Engineering, SE-1 pp. 59-67 (March 1975).

[2] B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys 4 pp. 97-147 (June 1972).

[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. #75CH0940-7CSR, 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 (Sept.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. # 75CH0940-7CSR, 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 (March 1975).

[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 (to appear)

[19] F. E. 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. 191-213 (December 1973).

[23] 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).

[24] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability", ACM Computing Surveys (to appear).

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

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

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

[28] 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).

[29] B. Elspas, M. W. Green, A. J. Korsak and P. J. Wong, "Solving Nonlinear Inequalities Associated with Computer Program Paths", Stanford Research Institute, Menlo Park, California; Preliminary Draft (October 1974).

[30] T. Cheatham, Talk presented to Department of Defense Invitational Conference on Software Verification and Validation, Syracuse, N. Y., August 1976.

[31] W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System", University of California, San Diego, Applied Physics and Information Science Department, Computer Science Technical Report #11, (May 1976).

[32] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw Hill, New York 1974.

[33] R. E. Griswold, J. F. Poage and I. P. Polonsky, The SNOBOL4 Programming Language, Prentice-Hall, Englewood Cliffs, N. J. Second Edition 1971.

[34] 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).

[35] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On Two Problems in the Generation of Program Test Paths" IEEE Transactions on Software Engineering SE-2, pp. 227-231 (Sept.1976).

[36] L. P. Deutsch, An Interactive Program Verifier, Ph.D.thesis, University of California, Berkeley (1973). Reprinted as technical report CSL-73-1, Xerox, Palo Alto Research Center, Pal Alto, CA (1976).

DISCUSSION OF SPECIFIC TASKS TO BE PERFORMED

The goal of this research will be to determine which testing capabilities should be incorporated into an ITS, and how these different capabilities can be optimally implemented and distributed among the three major testing subsystems. Much of the proposed work will be conceptual, focusing on the adaptation of existing algorithms to meet variout testing needs, and the creation of new algorithms. Another important aspect of the work will be experimental. As noted in the previous text, many of the primary goals of testing systems (such as the determination of the executability of an arbitrary path through an arbitrary program) are provably impossible, hence alternatives are proposed. Therefore an important aspect of this work will be to study the adequacy of various proposed heuristic approaches to demonstrably unsolvable problems. In particular, it is proposed that a prototype ITS be constructed, and that the ITS be designed for maximum flexibility through extensive modularity. In this way, different heuristic solutions can be implemented, experimented with, and evaluated. The following list enumerates various specific aspects of this overall project.

1. STUDY ERROR AND ANOMALY PHENOMENA WHICH CAN BE DETECTED BY DATA FLOW ANALYSIS

This task entails a search for those phenomena which can be detected by data flow analysis and which are either symptomatic of errors or constitute outright errors themselves. In the DAVE system, we have shown that data flow analysis techniques can be used to detect errors such as reference to undefined variables. In the above text, misuse of data types, references to items in unopened files, division by zero, array bounds violations, and improper use of semaphores in concurrent processing were cited as examples of other phenomena which could be detected in similar ways. This study would identify and categorize as many of these phenomena as possible. It is anticipated that the study would range over a wide variety of languages, including languages permitting recursion and parallelism, assembly languages, and microprogramming languages. The primary goal of this task is to identify the error phenomena which can be detected by data flow analysis, to discover good algorithms for their detection, and to characterize the limitations of this approach. The last aspect of this goal seems attainable only after algorithms for detecting several of these phenomena have been implemented and integrated into a working ITS. After such an implementation has been completed, it can be tested using a set of programs drawn from real-world programming environments. The adequacy of any heuristic approaches could then be determined and analyzed. Similarly, the actual division of labor between static analyzer and symbolic executor could be analyzed and evaluated.

2. STUDY ERROR AND ANOMALY PHENOMENA WHICH CAN BE DETECTED BY SYMBOLIC EXECUTION

This task entails a search for the phenomena which can be detected by symbolic execution and which are either symptomatic of errors or constitute outright errors themselves. Earlier work has shown that symbolic execution is a powerful tool for determining whether or not a given path is executable. In addition, it has been shown that symbolic execution can be used to check for the possibility of array bounds violations and division by zero through the introduction of automatically generated artificial constraints. The above text has also proposed that it is reasonable to consider the possibility of verifying assertions about the relations between the values of program variables through symbolic execution. The primary goal of this task is to identify the error phenomena which can be detected by symbolic execution, to propose techniques by which such detection can be effected, to characterize the limitations of this approach, and to evaluate the practical significance of these limitations. The last aspect of this goal seems attainable only after a symbolic execution system with some of these error detection capabilities has been built and integrated into an ITS. After this has been accomplished the necessary evaluations can be made based upon experience in analyzing a body of real-world programs.

3. EXPLORE METHODS FOR INHIBITING THE GENERATION OF UNEXECUTABLE PATHS DURING STATIC ANALYSIS

It has already been observed that symbolic execution is a powerful tool for identifying unexecutable paths. The above text also points out that this process is relatively expensive, and an unexecutability finding by the symbolic executor is awkward, as it leaves in doubt whether or not the phenomenon of interest along the given path is actually accessible by some executable path. All of this points to the desirability of being able to filter out some of the unexecutable paths during static analysis. Such a capability promises to be more cost effective and poses fewer problems to the overall structure and flow of capabilities within the ITS. The above text proposes a method by which data flow analysis can be used to filter out some unexecutable paths during static analysis. This method would be implemented and integrated into an ITS. Then its effectiveness in filtering out unexecutable paths in an actual set of real-world programs would be measured. Embellishments to this algorithm would be studied as indicated by actual experimental results. Other approaches to the filtering problem would also be investigated as part of this task.

4. STUDY THE APPLICABILITY OF LOOP INVARIANT SYNTHESIS TECHNIQUES TO THE DETECTION OF PATHS WHICH BEAR ERRORS AND ANOMALIES

The synthesis of loop invariants has already been studied in connection with formal proof techniques for program correctness. In the above text it was proposed that a symbolic execution system could be used to synthesize some of these invariants. The invariant could then be used, in conjunction with other information about the relationships between variables, to hypothesize the number of loop iterations which would force the occurrence of such errors as out of bounds referencing of an array. The purpose of this task is to study the feasibility of incorporating such a loop count synthesis capability into the symbolic execution. There are a number of aspects to this task. There must be a study of which error and anomaly phenomena might be detected in this way. There must also be a study of whether existing loop invariant synthesis techniques are adequate to produce the information needed to specify the paths bearing these errors. A related line of research would involve the production of new loop invariant synthesis capabilities as dictated by path synthesis needs. There is also a more experimental aspect to this work. Some loop iteration count synthesis capabilities would be built into a symbolic execution system and tested by examining a body of real-world programs. Such actual test cases should allow the evaluation of whether the synthesis of loop counts should be pursued further as an error detection capability and should also help dictate the direction of further research.

5. INVESTIGATE METHODOLOGIES FOR THE GENERATION OF PATHS WHICH ARE ALTERNATIVES TO THOSE WHICH ARE FOUND TO BE UNEXECUTABLE

As was observed above, it must be expected that the static analysis subsystem will at times generate paths which are unexecutable, although they bear errors or anomalies. When this occurs, the impossibility of the occurrence of the error has not been established, only the impossiblity of its occurring along the given path. It must be expected that the determination of unexecutability may be made by the symbolic execution subsystem, despite the presence of filtering mechanisms within the static analysis system. This task, therefore, is concerned with finding an adequate mechanism for generating an alternative path which bears a given error or anomaly after the error or anomaly has already been found to lie on one or more unexecutable paths. Information within the symbolic execution system's constraint solver should be helpful here, as it can be used to point out which conflicting constraints cause the unexecutability. Hence it seems that this path regeneration mechanism might be at least partially imbedded within the symbolic execution subsystem. An important aspect of this task is the development of a criterion which can be used to abort the search for an executable error or anomaly bearing path, because it is sometimes the case that no such path exists. Clearly, most of these questions are best explored experimentally. Hence this task requires the completion of an ITS in which the path regeneration capability would be carefully embodied in modules which could be altered and interchanged during experimentation.

6. STUDY METHODS FOR DIVIDING PATH SPECIFICATION RESPONSIBLITY BETWEEN THE STATIC ANALYSIS SUBSYSTEM AND THE SYMBOLIC EXECUTION SUBSYSTEM

Some of the foregoing task descriptions have made it clear that many difficulties arise if the static analysis subsystem is relied upon to furnish a complete specification of a path along which an error or anomaly occurs. It has been noted, for example, that unexecutable paths may be generated in this way. This causes difficulties in determining whether the underlying error or anomaly is actually executable along some other path and, in such cases, clearly implies the problem of determining and specifying one such path. It has also been noted that symbolic execution seems to be a promising vehicle for determining, in certain cases, loop iteration counts which force the occurrence of certain errors and anomalies. In these cases, static analysis seems ill equipped for determining this count. Hence it seems reasonable to study strategies for dividing between the static analysis and symbolic execution subsystems the responsibility for completely specifying anomaly and error bearing paths. The static analysis subsystem might be relied upon to produce only an abbreviated path specification, consisting of an ordered list of critical nodes to be included on the path, other nodes to be omitted from the path, and interesting loops to be iterated an unspecified number of times. The symbolic execution subsystem would then be expected to use its more detailed information about the program and path to determine the exact loop iteration counts to be used and the identities of the nodes to be included between the critical nodes. Various methodologies exploiting differing divisions of path generation labor would be defined and examined experimentally.

7. EXAMINE THE ADEQUACY OF CURRENT DATA FLOW ANALYSIS ROUTINES AND CONSTRUCT NEW ONES AS NEEDED

Anticipating that the first task described above will identify phenomena intrinsically different from those presently attacked very effectively by LIVE and AVAIL algorithms (see [24]), we propose to search for new algorithms which might be better suited to handle these phenomena. Such phenomena might be characterizable only by a more complex sequence of events than simply an alternation of two events such as is the case for LIVE and AVAIL (in these cases, the two events are GEN and KILL). We propose to explore new algorithms which are well suited for the detection of phenomena which can only be characterized by somewhat more complex patterns, perhaps involving more than two events. This work would include an investigation of the relation between the complexity of the pattern searched for and the complexity of the algorithm.

8. INVESTIGATE SUPERIOR METHODOLOGIES AND ALGORITHMS FOR SIMPLIFYING AND SOLVING CONSTRAINTS ARISING FROM PROGRAM PATH PREDICATES

It was noted in the above text that it is impossible to determine the executability of an arbitrary path through an arbitrary program, because this is tantamount to being able to determine the consistency of an arbitrary set of non-linear inequalities. This task is concerned with developing a methodology for simplifying and solving such systems of constraints which arise in the course of examining program paths. The simplification methodology should be based upon the existing literature on symbol manipulation, and adapted to best serve the needs arising from constraints embedded in actual code. The constraint solving methodology must be able to accept mixed systems of constraints (i.e. systems involving both integer and real values) as well as non-linear constraints. Such a methodology would be synthesized from the existing literature and technology, and augmented where necessary by original research. This task would necessarily be guided by experiences with different methodologies embedded within an ITS. Initially, an empirical study of programs, using existing symbolic execution systems, would be conducted to determine the sorts of constraint solving and simplification capabilities which appear to be most important. The results of this study would be used to guide in the development of improved methodologies.

9. INVESTIGATE PATTERNS OF ARRAY USAGE IN ORDER TO GUIDE THE DEVELOPMENT OF SPECIALIZED ARRAY ANALYSIS METHODS WITHIN THE SYMBOLIC EXECUTION SYSTEM

In the above text it was noted that certain patterns of array definition and reference cause problems of varying degrees of difficulty in attempting symbolic execution. Some work in the detection of parallelism in programs indicates that specialized forms of analysis can be employed to facilitate the examination of some of these patterns. The applicability of such techniques to symbolic execution would be studied. An empirical study of the patterns of array reference and definition within real-world programs would also be conducted in order to determine which of the troublesome patterns are most common in actual code. The results of this empirical study would then be used to guide efforts at adapting known analytic methods and devising new methods.

ESTIMATE OF DISTRIBUTION BETWEEN UNIVERSITIES
OF PERSONNEL EFFORT FOR PROPOSED TASKS

Task	person-years	University of Colorado share		University of Massachusetts share	
		%	# person-years	%	# person-years
1	6	100	6	0	0
2	4	25	1	75	3
3	2	100	2	0	0
4	2	50	1	50	1
5	2	50	1	50	1
6	2	50	1	50	1
7	3	100	3	0	0
8	3	67	2	33	1
9	.5	0	0	100	.5
Totals	24.5		17.0		7.5

ESTIMATED PROFILE OF PERSONNEL DISTRIBUTION
FOR TASKS TO BE PERFORMED AT CU

Task	Total person-years	Principal Investigators person-years	Other Investigators person-years	Programmer person-years	Research Assistant person-years
1	6.	.5	.25	1.75	3.5
2	1.	.1		.25	.65
3	2.	.25		.25	1.5
4	1.	.25	.25		.50
5	1.	.1		.25	.65
6	1.	.1		.25	.65
7	3.	.5	.5		2.0
8	2.	.25	.5	.25	1.0
9	0.				
Totals	17.	2.05	1.5	3.0	10.45

BUDGET

A. <u>Salaries and Wages</u>	Year 1	Year 2	Year 3
1. Co-Principal Investigators			
L. Osterweil			
10% time, 9 mos. A.Y.(1mm)	$ 1,800.	$ 1,950.	$ 2,150.
100% time, 2 mos. summer (2mm)	3,700.	4,050.	4,400.
L. Fosdick			
10% time, 9 mos. A.Y.(1mm)	3,300.	3,600.	3,900.
100% time, 2 mos. summer (2mm)	6,800.	7,400.	8,000.
2. Associate Investigator			
H. Gabow			
100% time 2 mos. summer (2mm)	3,500.	3,850.	4,200.
3. Research Associates			
To be named			
50% time 9 mos. A.Y.(4.5mm)	9,000.	10,000.	11,000.
4. Staff scientist			
100% time, 12 mos. (12mm)	15,000.	16,500.	18,000.
5. Secretary			
100% time, 12 mos. (12mm)	7,500.	8,000.	8,500.
6. Graduate Research Assistants (6)			
50% time, 9 mos. A.Y.(27mm)	27,000.	29,000.	31,000.
100% time, 2.5 mos. summer (15mm)	15,000.	16,000.	17,000.
TOTAL SALARIES AND WAGES	$ 92,600.	$100,350.	$108,150.

BUDGET (continued)

	Year 1	Year 2	Year 3
B. Fringe Benefits			
TIAA: 7% of faculty salaries	$ 1,400.	$ 1,500.	$ 1,600.
PERA: 10.64% of staff salaries	2,300.	2,500.	2,700.
TOTAL FRINGE BENEFITS	$ 3,700.	$ 4,000.	$ 4,300.
C. Permanent Equipment			
None			
D. Expendable Supplies and Equipment			
Office and file maintenance supplies	$ 2,000.	$ 2,000.	$ 2,000.
E. Travel			
Domestic:			
1. Project personnel to meetings and consultants to and from Boulder	$ 4,000.	$ 4,000.	$ 4,000.
2. Coordination trips to University of Massachusetts	1,000.	1,000.	1,000.
Foreign:			
PI's to organize and participate in IFIP activities		4,000.	2,000.
TOTAL TRAVEL	$ 5,000.	$ 9,000.	$ 7,000.
F. Publication Costs			
Page Costs, Reprints, Duplication	$ 2,500	$ 2,500.	$ 2,500.

BUDGET (continued)

		Year 1	Year 2	Year 3
G.	Other Direct Costs			
	Communication costs--including telephone computer access	$ 1,500.	$ 1,500.	$ 1,500.
	Equipment maintenance	2,000.	2,000.	2,000.
	Consultants' Fees: (20 days/year @$150.)	3,000.	3,000.	3,000.
	Computer Costs:			
	On CU machine	30,000.	30,000.	30,000.
	Rental on other machines	3,000.	3,000.	3,000.
	TOTAL OTHER DIRECT COSTS	$ 39,500.	$ 39,500.	$ 39,500.
H.	Total Direct Costs	$145,300.	$157,350.	$163,450.
I.	Indirect Costs			
	On Campus: 53% of Salaries and Wages	$ 49,600.	$ 53,200.	$ 57,300.
	GRAND TOTAL:	$194,900	$210,550.	$220,750.

BUDGET NOTES

The following section supplies elaboration and explanation of some of the expenditures itemized in the budget for this proposal.

1. It is important to note that the responsibility for the integration of all programs produced under this proposed work into a single operational system will be assumed by the University of Colorado. Distribution and maintenance will likewise be performed by the University of Colorado. Hence the University of Colorado requests considerable support for these efforts both in personnel and in computer time. Among the personnel requested is a full time staff scientist who will have overall responsibility for the integration, dissemination and maintenance of the entire system as it evolves. Due to the size and complexity of the proposed system, such a person is deemed crucial to the success of the project.

2. The attached charts provide additional detail as to how the various proposed personnel will be utilized in completing the proposed individual tasks.

3. Because the proposed work will require expertise in a wide variety of areas, some associate investigators and other investigators are proposed. In particular, Professor Harold Gabow is proposed as an associate investigator because of his expertise in the design and analysis of algorithms for data flow analysis. The expertise of other investigators will also be useful in performing such tasks as designing the non-linear inequality solving capability for the symbolic execution subsystem and building a capability for synthesizing and utilizing loop invariants.

4. The attached budget includes funding for a full time secretarial position. This request is made because past experience in administering grants of similar size has demonstrated that considerable secretarial work is necessary to the smooth functioning of the research activities. The proposed secretarial position would entail responsibility for such tasks as typing and duplicating reports, performing accounting and auditing of computer usage and general project funds, handling the considerable correspondence that seems likely to be necessary and facilitating coordination between the University of Colorado and the University of Massachusetts.

5. The attached budget requests considerable travel funds. Our past experience indicates that substantial travel will be required during the course of the project. The travel funds would be used to support trips to scholarly conferences, trips to user sites to install and maintain the software produced by this project, and travel by outside consultants visiting our project for short periods of time. We have also requested sufficient travel to cover the cost of several trips to the University of Massachusetts

for the purpose of coordination. We anticipate that some of the work done by the University of Colorado personnel will have to be performed at the University of Massachusetts in order to coordinate effort most efficiently. Hence adequate travel funds must be allowed for.

6. We have also requested foreign travel funds due largely to the involvement of one of the principal investigators with IFIP working group WG 2.5. This participation will entail a trip to Europe during year 3 of the proposed grant period to attend a group meeting, and another trip during year 2 in order to organize and attend an IFIP Working Conference. It is requested that sufficient travel funds be allocated to allow the other principal investigator to attend this working conference, as the topic of the conference relates to quality software development, and is thus closely related to the topic of this research proposal.

7. The attached budget includes a request for substantial communications funding. This is partially due to the fact that we anticipate a large amount of communications between the two universities will be carried out via telephone, where the telephone will be used both for voice communication, and for accessing computer files. We hope that much of the software coordination work will be carried out by telephone data links, thereby reducing to some extent the need for coordination trips.

8. The proposed maintenance budget is designed to include funds for the maintenance of terminal equipment already owned by the University of Colorado, but intended for heavy use by the proposed project. The maintenance expense proposed here also includes an allocation to the maintenance of a microprogrammable computer. Earlier in the technical proposal, we stated that some of the data flow analysis techniques to be explored appear to have strong applicability to the analysis of microcode. We are, in a separate proposal, requesting funds for the purchase of a microprogrammable computer in conjunction with a number of other investigators. These others would use the proposed machine in a variety of ways, thus generating a body of microcode which we could then use to gauge the effectiveness of our techniques at detecting and validating for errors. We intend to share the maintenance cost and are requesting funds for our share, in anticipation of the possibility that funding for the purchase of the machine will be forthcoming.

9. In the attached budget, we have requested funding for bringing consultants to Boulder. In previous grants of this magnitude we have had such funding, and have found that the specialized expertise of the people who have visited us has been most valuable. In this proposed work, there will be an increased emphasis on integrating widely diverse capabilities. Hence, here even more than in the past, we would like to be guided by experts in divergent fields for short periods of time in order to be sure that our coordination and integration efforts are not based on faulty assumptions.

10. We have requested a special allocation of computer funds to be expended on machines other than those available at the University of Colorado. These funds would be used to compensate computing facilities which we might use to test implementations of our software on different types of hardware and operating systems. In the past we have relied upon the hospitality of friends to provide us with free computer time and services for the purpose of testing our implementations, and thereby verifying the portability of our software. We would prefer to be able to provide our own support for such efforts in the future.