A STRATEGY FOR EFFECTIVE INTEGRATION OF VERIFICATION AND TESTING TECHNIQUES†

by

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

CU-CS-181-80 July, 1980

†This is a preliminary version of a paper to appear in the Handbook on Software Engineering, C. R. Vick and C. V. Ramamoorthy, editors, to be published by Van Nostrand Reinhold Company. It is appearing in this format solely to provide early circulation of these ideas.

*This work was partially supported by grant number MCS77-02194 from the National Science Foundation and grants DAAG29-78-G-0046 and DAAG29-80-C-0094 from the U.S. Army Research Office.

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.

CONTENTS

I. Introduction..........................2

II. Dynamic Testing.......................6

III. Static Analysis......................15

IV. Formal Functional Analysis...........20

V. The Integration Strategy.............29

VI. Software Lifecycle Considerations....34

VII. References...........................39

I. INTRODUCTION

The purpose of this paper is to convey to the reader a sense of how the wide variety of approaches to testing, analysis, verification and validation of software can be combined by a single overall strategy. The goal of this strategy is to enable the user to incrementally raise his confidence in his software through incremental expenditures of effort and resources. This is to be done by employing the various tools and techniques currently and foreseeably available in an orderly systematic way which exploits their complementary patterns of strength and weakness.

In order to best appreciate the integration strategy it is important to first understand the rationale for taking the goal of confidence raising as being of central importance. It is becoming widely agreed and accepted that software is best thought of as a product. As such, its production must be carefully managed through a succession of stages. This succession of stages is commonly referred to as the software production lifecycle. A primary goal of establishing such a lifecycle is to establish interim phases of the production process and the products of these intermediate phases. It is through examination of these intermediate products that management should be able to determine whether the production process is proceeding satisfactorally, and if not determine the appropriate remedial actions.

The ability of management to observe, interpret and evaluate all the products (intermediate and final) of the software production process is generally regarded as being the pivotal capability for the success of a software project. Correspondingly, there has been a great deal of work done on creating various lifecycle models, rigorizing and subdividing the various lifecycle phases, defining intermediate products, and devising management guidelines. These and related topics are dealt with elsewhere in this volume. In this paper we deal with the issue of raising confidence in software products, which we claim is central in importance to the goal of effective management.

A major purpose in subdividing the production process into phases is to create intermediate points in the process for the purposes of monitoring and review. These monitoring, or review, activities have as their goal the determination of whether or not current progress has been sufficient to warrant proceeding to the next lifecycle phase (or subphase). The review process which occurs after coding (the final lifecycle production phase) is usually called acceptance testing, and is regarded by some people to be an entire lifecycle phase by itself.

In any event, its purpose is to enable a decision as to whether progress to date has been sufficient to justify release of the finished product. In this essential sense, this review can be seen to be similar in nature to all other reviews — having as its goal the raising of the confidence of the reviewers in the product or products at hand.

Thus in a very important sense it is reasonable to say that all progress through the software development process is predicated upon our ability to gain confidence in our completed products. This confidence is gained in many ways, and in different ways by different people. Testing, analysis and verification procedures are used. Automated tools, manual procedures, and informal means are employed. Formal organizational procedures and totally informal means are employed. Unfortunately there is little understanding of the inherent value of most of these procedures, and little appreciation of how these diverse approaches might be integrated to enable systematic, incremental confidence raising. Hence very important decisions are often based upon confidence which is badly placed, or worse yet at a regrettably low level.

Clearly what is needed is an orderly systematic procedure for raising confidence incrementally at incremental cost of effort and resources. It must be understood at the outset that there are no absolutes in this process. Confidence, as a human emotion, cannot be uniformly and unerringly described and manipulated. Different individuals under different sets of circumstances become more or less confident in response to identical procedures. Moreover, different individuals under different sets of circumstances set different threshold levels as being sufficient to declare that progress to date has been "sufficient" or "adequate." The procedure outlined in this paper attempts to take these difficulties into account. It offers no ways of obtaining absolute assurances, but rather a coordinated set of ways in which confidence can be raised in various aspects of various software products. As such it should be a highly useful procedure to software decision makers - be they coders, analysts or managers.

The notion of confidence is inextricably tied in with the notion of an error. Confidence in a software product decreases with discovery of errors and increases as assurances of the absence of errors are given. Hence our integration strategy focusses on being able to find and eliminate errors, and then progressively rule out the possibility of errors of various kinds. With errors being of such central importance, it is necessary to have a firm notion of what an error is.

We define an error to be a failure of an algorithmic process (e.g. a body of code) to perform as expected. Thus our integrated strategy is directed sharply at detecting and disproving deviations of solutions from intent. What is most striking is that the process of detecting and disproving errors is heavily dependent upon knowing the intent of a solution, and yet this intent is rarely specified in any rigorous form. It is crucial to observe that in the absence of complete, formal specifications of intent, error detection and disproof are doomed to also be incomplete, informal, and inadequate bases for meaningful confidence incrementation. Thus a key component of our strategy is the incorporation of a capability for allowing the user to incrementally supply specifications of intent as the bases for error detection and disproof. These specifications of intent are to be supplied in the form of assertions — predicates, whose value, the user claims, is always true. By allowing different users to supply their own assertions we believe we have devised a mechanism whereby different users can, at different times, each effectively pursue the raising of their own confidence levels to their own desired confidence thresholds.

Before we can effectively promulgate this integrated strategy it is necessary to first describe the major classes of tools and techniques to be incorporated. In the following sections, we present these classes of techniques and specify the strengths and weaknesses of each in being able to discover and/or disprove adherence of solution to specifications of intent. Most of these techniques have been devised and developed with the aim of facilitating confidence improvement for code. In a later section we shall show that these techniques are also useful in increasing confidence in intermediate software products such as requirements and design specifications. We believe, however, that their inherent capabilities can best be presented, compared, and contrasted in the context of their most familiar application — namely to code. Thus, to facilitate the presentation of these capabilities, they will all be applied to a single example FORTRAN program. This program, shown in Figure 1, is somewhat contrived to appear plausible, yet contain a variety of errors.

The purpose of this program is to compute, store, and print out the costs of covering a sequence of areas with some covering materials. The program reads in N, the number of costs to be computed, initializes pi, and then enters the primary iteration DO loop. Inside this loop, the program reads in PSF, the cost per square foot of the material; LCRT, an integer used to denote whether the area is a rectangle (if LCRT is 1), a circle (if LCRT is 2), or a triangle

Statement #

```
                         COMMON/B/AREA  (10), COST (10)
 1                       READ (5,1) N
 2                       PI=3.1416
 3                       DO 200 I=1, N
 4                       READ (5,2) PSF, LCRT, D1, D2
 5                       IF (LCRT.NE.2) TO TO 10
 6                       AREA (I)=AREAR (D1, D2)
 7                       GO TO 100
 8          10           IF (LCRT.NE.2) GO TO 20
 9                       AREA (I)=AREAC (P,D1)
10                       GO TO 100
11          20           CALL AREAT (D1, D2, AREA (I))
12         100           CALL DOLS (PSF, I)
13                       WRITE (6,3) COST (I)
14         200           CONTINUE
15                       STOP
             1           FORMAT (12)
             2           FORMAT (F6.2, I2, 2F10.4)
             3           FORMAT (1H, F8.2)
                         END
16                       FUNCTION AREAR (A,B)
17                       AREAR=A*B
18                       RETURN
                         END
19                       FUNCTION AREA C(PI,RAD)
20                       AREAC=PI*RAD**2
21                       RETURN
                         END
22                       SUBROUTINE AREAT (B,H, AREA)
23                       AREAT=0.5*B*H
24                       RETURN
                         END
25                       SUBROUTINE DOLS (PSF,I)
                         COMMON/B/COST (10), AREA (10)
26                       COST (I)=PSF*AREA (I)
27                       RETURN
                         END
```

Figure 1. A program containing some errors designed to show the relative capabilities of the techniques discussed in this chapter.

(if LCRT is 3); and D1 and D2, the two dimensions of the area (D2 is unused if LCRT is 2). The program then branches on LCRT to three different subprograms, AREAR, AREAC, and AREAT, which are supposed to compute the area of the rectangle, circle or triangle (respectively), and place the value of this area in the array location AREA (I). Subroutine DOLS is then called to compute COST(I), the product of AREA(I) and PSF. Finally COST(I), the desired result, is printed out.

The program then returns to the READ statement to obtain new values for PSF, LCRT, D1 and D2. This loop is iterated N times at which time the iteration stops and program execution terminates.

Close inspection of the program reveals that it contains errors, some of which are not very obvious. Perhaps the most obvious error is that the value of pi is set into the variable PI, but the variable P is used to pass this value into AREAC, the subprogram which requires it. A second error is that there is a misspelling in the subroutine AREAT. The third parameter is named AREA, but the body of the subroutine defines a value for the variable AREAT instead. Hence upon return there is no value given to the main program variable AREA, which is referenced in a subsequent computation. A third error involves the COMMON block B, which is used for communication between the main program and DOLS. B contains the variables AREA and COST. DOLS, which expects AREA to contain the computed area, uses it to compute the value of COST, which is then passed through B back to the main program. Unfortunately, the order of declaration of AREA and COST in the main program is the reverse of the order of declaration in DOLS. A fourth error is found in statement 4. The test of LCRT .NE.2 is incorrect — it should be LCRT.NE.1. Because of this, LCRT is not correctly used as a switch.

We shall now proceed to show how each of four different classes of techniques is able or unable to detect or disprove these and other types of errors in this specimen piece of code.

II. DYNAMIC TESTING

The term dynamic testing, as used here, is intended to apply to most of the systems known as execution monitors, software monitors, and dynamic debugging systems (see for example [1,2,3,4]). The term dynamic testing is used because in contemporary usage it has come to suggest the most important feature of this technique.

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 statement just executed, the names of those variables whose values have 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 tables and variable evolution trees.

Suppose that the following inputs are read in by the example program in Figure 1:

Input Record #				
	(N)			
1	3			
	(PSF	LCRT	D1	D2)
2	2.00	1	3.0	4.0
3	3.50	2	5.0	
4	4.50	3	2.0	5.0

Following is an execution history which is typical of what would be created by a dynamic testing system such as [1,2,3,4]:

Execution #	Statement #	Type	Variables Used as Inputs and Their Values	Variables Used as Outputs and Their Values	Next Stmt. Exec.	Outcome if an IF Statement
1	1	READ		N:3	2	
2	2	Assignment		PI:3.1416	3	
3	3	DO	N:3	I:1	4	
4	4	READ		PSF:2.00 LCRT:1 D1:3.0 D2:4.0	5	
5	5	IF	LCRT:1		8	TRUE
6	8	IF	LCRT:1		11	TRUE
7	11	CALL	D1:3.0 D2:4.0 I:1 AREA(1):*	Set upon return	22	
8	22	Subroutine			23	
9	23	Assignment	B:3.0 H:4.0	AREAT:6.0	24	
10	24	RETURN			11	

(continued)

Execution #	Statement #	Type	Variables Used as Inputs and Their Values	Variables Used as Outputs and Their Values	Next Stmt. Exec.	Outcome if an IF Statement
11	11	CALL(Value return)		D1:3.0 D2:4.0	12	
12	12	CALL	PSF:2.00 I:1	AREA(1):*	25	
13	25	SUBROUTINE			26	
14	26	Assignment	PSF:2.00 I:1 AREA(1):*	COST(1):*	27	
15	27	RETURN			12	
16	12	CALL(Value return)		PSF:2.00	13	
17	13	WRITE	COST(1):*		14	
18	14	CONTINUE	I:1	I:2	4	
19	4	READ		PSF:3.50 LCRT:2 D1:5.0 D2:*	5	
20	5	IF	LCRT:2		6	FALSE
21	6	Assignment	D1:5.0 D2:*	Function Invoked	16	
22	16	FUNCTION Definition			17	
23	17	Assignment	A:5.0 B:*	AREAR:*	18	
24	18	RETURN			6	
25	6	Assignment Function Return	AREAR:* I:2	AREA(2):*	7	
26	7	GO TO			12	
27	12	CALL	PSF:3.50 I:2	Subroutine invoked	25	
28	25	SUBROUTINE			26	
29	26	Assignment	PSF:3.50 AREA(2):* I:2	COST(2):*	27	
30	27	RETURN			12	
31	12	CALL(Value return)		AREA(2):*	13	
32	13	WRITE	COST(2):*		14	
33	14	CONTINUE	I:2	I:3	4	
34	4	READ		PSF:4.50 LCRT:3 D1:2.0 D2:5.0	5	
35	5	IF	LCRT:3		8	TRUE
36	8	IF	LCRT:3		11	TRUE
37	11	CALL	D1:2.0 D2:5.0 I:3 AREA(3)*	Subroutine invoked	22	

(continued)

Execution #	Statement #	Type	Variables Used as Inputs and Their Values	Variables Used as Outputs and Their Values	Next Stmt. Exec.	Outcome if an IF Statement
38	22	SUBROUTINE Definition			23	
39	23	Assignment	B:2.0 H:5.0	AREAT:5.0	24	
40	24	RETURN			11	
41	11	CALL Value Return		AREA(3):*	12	
42	12	CALL	I:3 PSF:4.50	Subroutine invoked	25	
43	25	SUBROUTINE			26	
44	26	Assignment	I:3 PSF:4.50 AREA(3):*	COST(3):*	27	
45	27	RETURN			12	
46	12	CALL(Value return)		AREA(3):*	13	
47	13	WRITE	COST(3):*		14	
48	14	CONTINUE	I:3	I:*	15	
49	15	STOP				

*This value is undefined. The dynamic testing system will record a specific value, namely the one assigned by the execution environment.

The previous table shows the wealth of detail captured in the execution history. It also indicates a primary problem in using the dynamic testing technique for error detection. Erroneous values are concealed amid a plethora of other information, complicating their discovery. It should be noted that the asterisks in the previous table indicate undefined values, but specific numbers (presumably incorrect) will appear in the actual output from a dynamic test. For this reason, summary statistics and tables are important products of dynamic testing executions.

A common summary table is the statement execution frequency table. For the previous execution this table would be:

Statement #	Number of Executions
1	1
2	1
3	1
4	3
5	3
6	2
7	1
8	2
9	0
10	0

(continued)	Statement #	Number of Executions
	11	4
	12	6
	13	3
	14	3
	15	1
	16	1
	17	1
	18	1
	19	0
	20	0
	21	0
	22	2
	23	2
	24	2
	25	3
	26	3
	27	3

This table shows which statements were heavily used in the test execution. This is important in attempting to improve the program's efficiency. From an error detection point of view, it is more significant to note that some statements (9, 10, 19, 20, 21) were never executed at all. These statements are the ones involved in computing the area of a circle. The test data is clearly intended to exercise this code. Hence, an error is indicated. Examination of the program's control flow should pinpoint the error in statement 5.

A variable evolution tree can also be useful in error detection. The variable evolution tree helps to indicate how data values are created out of program inputs and other data values. Suppose that the value printed by statement 13 at execution sequence number 47 is recognized as being incorrect. The source of the erroneous value can be found more readily by determining how the value was created. In this case, many dynamic testing systems are able to determine the computation which produced any value evolved by the program. The sources of all values of all variables used in that computation can likewise be determined. This process can be continued until constant values and read-in values are reached. This entire record can in some systems be summarized and displayed as a tree structure.

In the example, the source of the value printed at execution sequence number 47 does not exist. This is a clear indication that the value, supposedly generated by subroutine DOLS is not emerging as expected. Examination of the variable evolution tree for the value of AREAT at execution sequence number 39 is helpful. The dynamic testing system shows AREAT received its value at sequence number 39

as the result of multiplying B by H. In sequence number 38, B and H were associated with variables D1 and D2 respectively. These, in turn, were initialized by a READ statement at sequence number 34. This evolution sequence is correct. Thus, the workings of the value passing from AREAT to the main program must be suspected, as well as the communications between DOLS and the main program. Both are in fact incorrect.

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, out-of-bounds array references and references to uninitialized variables. The monitors implanted are usually programmed to automatically issue error messages to a special file immediately upon detecting such conditions in order to avoid having the errors concealed by the bulk of a large execution history.

Thus, for example, most dynamic testers automatically insert error monitoring code into the program being tested. Before a division is attempted, this monitoring code checks the divisor for zero. If it is zero, a message is generated and the division is averted. This avoids loss of execution control due to seizure by the execution environment. Similarly, before a subscripting operation is attempted, monitoring code will (at the user's option) check the subscript value to see if it is within the range declared for the array dimension being subscripted. In many dynamic testing systems, all variables are initialized to a special value which is unlikely or impossible to arise in an actual computation. Before any value is used in a computation, it is compared for equality to this special value. If the comparison succeeds, then the computation being attempted is referencing an undefined value. Monitoring such as this will identify the presence of undefined references in execution sequence numbers 14, 17, 21, 23, 25, 29, 32, 44 and 47.

Some systems [2,3] also allow the tester to use assertions to create his own monitors, direct their implantation anywhere within the program, and specify where and how their messages are to be displayed. The greatest power of these systems derives from the possibility of using them to directly determine whether a program execution is proceeding in accordance with explicit formal statements of intent. The intent of the program is expressed formally by the tester, using sets of assertions about the desired and/or correct relation between values of program variables. These assertions may be specified to be of local or global

validity. The dynamic testing system then creates and places monitors as necessary to determine whether the program is behaving in accordance with asserted intent as execution proceeds.

Thus, for example, it would be useful to assert that at source statement number 13, the predicate

```
COST(I) = AREA(I) * PSF
```

is always true.

In most actual dynamic testing systems this is done by inserting a statement of the form

```
C ASSERT COST(I) = AREA(I) * PSF
```

immediately after statement 13. This statement is a Fortran comment, and is hence ignored by usual compilers; serving only the useful purpose of being good code documentation. The statement is, however, recognizable by a probe insertion preprocessor, because it begins with

```
C ASSERT
```

The preprocessor converts this comment statement into the executable statement

```
IF(COST(I).NE.AREA(I) * PSF) report violation
```

and places it after statement 13 in the source text stream to be directed to a conventional compiler. A diagram of the information flows in a dynamic testing system is shown in Figure 2.

Similarly, the assertion

```
C ASSERT AREA(I) = 3.1416* D1**2
```

inserted between statements 9 and 10 will help identify the misspelling in statement 9. It is important to observe that this error will not be noticed at all until the error in statement 5 is corrected, allowing statement 9 to be executed. This indicates a significant weakness in the dynamic testing technique, which will be elaborated upon shortly.

The preceding two examples are of local assertions — assertions whose validity is checked only at the point of specification of the assertion. The ability to create global assertions is a more powerful capability. A global assertion is one whose validity is asserted (and is, therefore, to be checked) over a specified range or extent within the source text. Most usually, one specifies an assertion to be valid throughout the execution of an entire procedure. This is specified by an assertion of form such as:

```
C ASSERT GLOBAL procname assertion.
```

It is worthwhile to note that array subscript checking is a specific instance of this type of assertion. It is equivalent to inserting in each

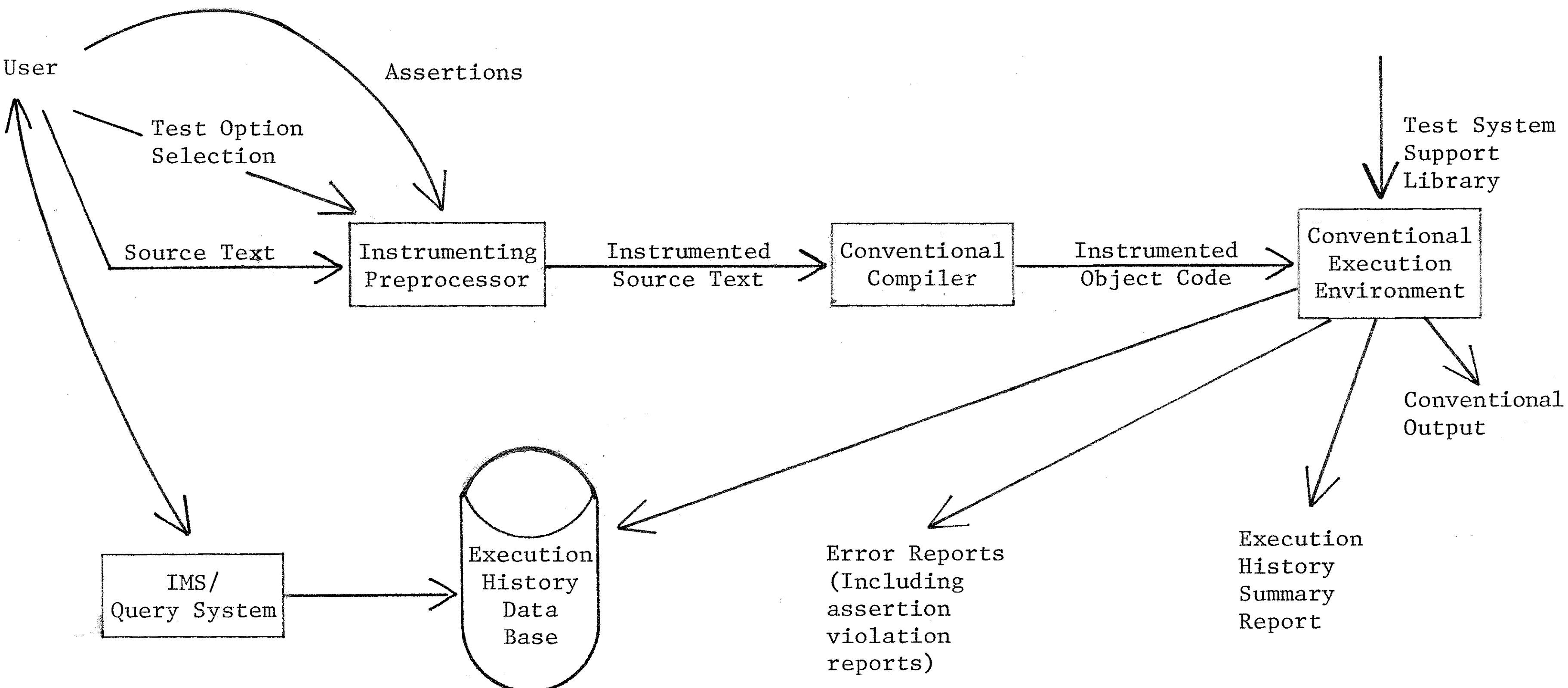

Figure 2. Data Flows in a Dynamic Testing System

program unit of our example program an assertion such as:

C ASSERT GLOBAL pgmunit $1 \leq$ SUBSCRIPT (AREA, COST) ≤ 10

Because this is such a frequently made request, most dynamic testing systems offer it as an option specifiable without the need for assertions. With the full assertion capability, however, it is possible to check any subscript, indeed any named variable, to see whether it remains within any range of values, over any specified body of code. Thus, for example, it would be useful to

C ASSERT GLOBAL mainpgm $1 \leq I \leq N$

It should be clear that such assertions require the generation of testing probe code at arbitrarily many places within the source text. In the array subscript range checking examples, there would probably be a probe placed before execution of every reference to an array being monitored. In simple variable range checking, there would have to be a probe after every alteration of the simple variable.

Often it is useful to create assertions specifying which procedure parameters are to be changed by the procedure execution and which are to remain unchanged. In particular an assertion of the form:

```
C ASSERT GLOBAL AREAT ALTER(AREA),
C                     STET(B,H)
```

placed after line 22 of our example, would have directly indicated an important flaw in AREAT. An assertion of this type requires saving the incoming argument values for later comparison to corresponding values upon execution of a RETURN.

The previous paragraphs have made it clear that dynamic testing systems have strong error detection and exploration capabilities. They excel at detecting errors during the execution of a program, and also at tracing these errors to their sources. These systems are capable of examining only a single execution of a program, however, and the results obtained are not applicable to any other execution of the program. Hence, the non-occurrence of errors in a given execution does not guarantee their absence from the program itself. It is thus seen that dynamic testing systems offer strong capabilities for detecting the presence of virtually any error which the user might specify. They have, however, no inherent capabilities for showing the absence of errors. Hence there is an important limitation on their ability to raise confidence, strongly suggesting the desirability of coupling them with other, complementary techniques.

It should also be clear that the wealth of detailed execution information which is responsible for much of the power of this technique is obtained only as

a result of an execution occurring in response to actual program input data. It is important to point out that this information is obtained at the cost of (often considerable) increases in execution time, and source code size. Furthermore, the generation of input data is the responsibility of the tester, and in many cases involves quite a significant amount of effort and insight into the program. It is important to recall in this context, moreover, that a significant amount of human involvement is also required in order to effectively use the capabilities of the system to detect and explore execution errors. Hence, it is seen that the power to obtain detailed insight comes in part from significant involvement of the human program tester, and is strongly predicated upon his close familiarity with the potential flaws in the program. Thus it is highly desirable that the human tester's efforts be somehow directed to the exploration of program phenomena which are certain or likely sources of error. Here too, the use of other tools and techniques should be expected to be of help.

Thus, summarizing, dynamic testing systems provide strong error recognition and exploration capabilities, but are unable to determine the absence of errors. Their results are narrowly applicable, being valid only for a single program execution. These results are quite extensive and detailed, however, providing sufficient material for deep insight. 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 knowledgable about the program being tested. The efforts of this human can be most effectively utilized if they are somehow steered in the direction of interesting and revealing phenomena.

III. 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 statement-by-statement 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. None of the errors in the program in Figure 1 can be detected by this type of scan, but this technique could identify as either errors or causes for concern each of the following variants of statements in Figure 1:

```
 2        PI = 3.14.6
 3        DO 200 I = 0,N
 4        READ(5,2)
17        AREAR = A * 1
23        AREAT = 0. * B * H
```

It should be noted that some of these statements are found erroneous by the syntax scanner of any conventional compiler. Others are not syntactically incorrect, but either violate semantic rules or strongly suggest the presence or symptom of an error. These could be found by statement-by-statement analysis. Compilers could readily detect such errors, but very few do so. There are stand-alone static analyzers which perform such analysis.

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 some conventional compilers. Other types, such as argument/parameter list agreement

are far less common in compilers, but are found in stand-alone static analysis systems [5,6]. None of the errors in the program in Figure 1 can be detected by this type of static analysis, but this technique could identify that errors are caused by each of the following variants of statements in Figure 1:

6	AREA(I) = AREAR(D1)
9	AREA(I) = AREAC(D1)
13	WRITE(6,3)COST(I,I)
19	INTEGER PI
25	COMMON/BB/COST(10),AREA(10)

The use of static analysis techniques to examine sequences of program events enables the detection of still more subtle types of program errors. In [7] 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 consists of examination of sequences of statement executions which are possible given the program's control flow structure, and determination of 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.

This type of static analysis is often called data flow analysis. It can be used to detect manifestations of most of the errors in the program in Figure 1. The misspelling of the variable PI would be indirectly detected by data flow analysis in two ways. First, a scan of the main program will show that PI is defined (receives a value) at statement number 2, but is never subsequently referenced. Second, the variable P will be found to be referenced before any definition. This latter conclusion will be reached after a scan of AREAC has been made, showing the first parameter to be used as an input (reference) by AREAC. Thus, it is known that P in statement 9 is used as a reference. Neither of these two diagnostics directly states that a misspelling has occurred, but both strongly indicate the error.

In a similar way, the error in the calling sequence of SUBROUTINE AREAT can be indirectly detected by static data flow analysis. The first indication of the error is that the third parameter of AREAT (namely AREA) is used neither as an input nor an output. The second indication is that the <u>local</u> variable AREAT is defined at statement 23, but never subsequently referenced. These two

diagnostics indicate that subroutine AREAT is generating a value which is never used, and that one of its parameters is not being used for interprocedural communication. It is not difficult to notice the variable naming mismatch based upon these indications.

This same type data flow analysis facilitates the detection of the COMMON list inversion between the main program and SUBROUTINE DOLS. The first array in COMMON in DOLS (called COST) is used as output, while the second array (called AREA) is used as input. Yet in the main program the second array (called COST) is used as an input following the invocation of DOLS, while the first array (called AREA) is defined immediately prior to the invocation of DOLS. Thus, the first array in COMMON is defined in the main program. Then DOLS is invoked and it is defined again without an intervening reference. The second array in COMMON is never defined in the main program at all. When DOLS is invoked, it is referenced, and upon return from DOLS it is referenced again. The patterns of reference and definition for both arrays are anomalous. Static data flow analysis will detect this anomalous behavior and focus attention on its cause. The inverted order of the two lists should then become apparent.

The DAVE static data flow analysis system [7] carries out such scans and produces messages such as described here. Systems such as this can use search techniques first developed in connection with program optimization [8,9,10,11] to carry out these scans in a highly parallel fashion. The patterns of reference and definition of all variables along all paths can be scanned for errors essentially in parallel. Hence, the existence of data flow errors such as described here can be detected or their presence can be disproven by these algorithms.

This capability is nicely complementary with the capabilities offered by dynamic analysis. This is perhaps best seen by observing that the static analysis capabilities just described (especially static data flow analysis) are in some cases capable of demonstrating the possibility or impossibility of violating assertions. In being able to show the impossibility of assertion violations, an improvement on the power of dynamic testing is offered, without the necessity for test executions.

For example static data flow analysis can readily verify the assertion

```
C ASSERT GLOBAL AREAT STET(B,H)
```

and readily be used to disprove

```
C ASSERT GLOBAL AREAT ALTER(AREA)
```

Static data flow analysis could also be used to show that it is impossible

to ever violate the assertion

C ASSERT GLOBAL mainpgm $1 \leq$ SUBSCRIPT (AREA, COST) $\leq$ N

This follows from the fact that all subscript references are through the variable I, whose range is regulated between 1 and N by statement 3. This is readily discernable by a static data flow analyzer.

Is is worthwhile to note that static analysis could do little to verify the assertion

C ASSERT GLOBAL mainpgm $1 \leq$ SUBSCRIPT (AREA, COST) ≤ 10

(i.e. the assertion that array bounds are never violated). This is because the array accesses never exceed the value of N, which is read in as an essentially unregulated value. If the statement

```
1.5      IF(N.GT.10)STOP
```

were inserted after statement 1, then static analysis could report the inviolability of the assertion. Without this check, however, the assertion's correctness could not be established. This in itself indicates the desirability of subsequent dynamic testing aimed at exercising this assertion. Hence we see how static analysis can be used to focus dynamic testing efforts. Static analysis is being used to demonstrate the lack of value or redundancy of certain assertion-generated dynamic test probes. Where this is possible the probes should be removed, thereby reducing the size, running time, and cost of the dynamic test execution without loss of analytic power. Where this is not possible dynamic testing attention and power should thereby be focussed on more probable sources of error and concern.

It can be seen from the preceding paragraphs that static analysis techniques offer capabilities for detecting and disproving the existence of deviations from intent. This is done by discarding a great deal of detailed information about 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 narrow aspects of a program's structure and functioning, but the results of this analysis are comprehensive and broadly applicable to all possible executions of the program. These capabilities are obtained without the need for human input or interaction. Efficient algorithms are used, generally making the cost of this analysis rather modest. A human tester is required, however, in order to interpret the results of the analysis and pinpoint the root sources of 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.

Probably the greatest weakness of static analysis, particularly data flow analysis, is its inability to examine the functional behavior of a program, and hence detect deviations of this behavior from intent. This renders static analysis useless as a tool for studying assertions of functional equality such as

```
C ASSERT COST(I) = AREA(I) * PSF
```

and

```
C ASSERT AREA(I) = 3.1416 * D1 ** 2
```

which were discussed earlier.

Because this type of assertion is one of the most important vehicles for raising confidence in a program, tools and techniques for studying it are extremely important. Dynamic testing can be used to detect the occurrence of violations, but in addition, there is a need for techniques and tools to demonstrate the impossibility of violating such assertions. Such techniques and tools are discussed in the next section.

IV. FORMAL FUNCTIONAL ANALYSIS

In formal functional analysis, the code comprising a program or algorithmic specification is compared to the functional intent of the program or specification, as captured and expressed in the form of assertions. The goal of this analysis is to prove theorems stating that the algorithmic specification actually achieves the intended functional behavior. Thus, both the assertions of intent and expressions of actual functioning must be phrased in rigorous mathematics.

This section is divided into two subsections. The first describes a technique for mathematizing the functional effect of a program. The second describes a technique for comparing this effect to a mathematical statement of intent in such a way as to comprise a rigorous proof of a desired theorem. A principal reason for this subdivision is that the technique for mathematizing the effect of a program, symbolic execution, has become an important technique in itself. Symbolic execution was originally thought of as a necessary adjunct, but has recently been recognized as a significant independent testing, analysis and verification technique. Hence, it will first be described from this point of view.

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 of variables are functions of constants and these input values and, thereofre, can be removed by substitution. The formulas can be examined by a human tester to see whether they embody the intent of the program, or, more important, they can be automatically compared to formal assertions of functional intent. If they are consistent with these assertions, 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]. The following indicates how a typical system can be used to determine the computational effect of paths through the program in Figure 1.

Consider the program path, 1, 2, 3, 4, 5, 8, 11, 22, 23, 24, 11, 12, 25, 26, 27, 12, 13, 14, 15. A symbolic execution of this path will keep track of the functional relation of every program variable to the program inputs values at every point along the path. The input values will be denoted by Roman numerals for the sake of clarity, and the functional relations involving these values will be expressed as formulas. The following table shows which variables are reset as each statement is executed and also shows the resulting functional relations of these variables to the input values. Thus, the current values of any variable at any given statement execution can be found by searching backwards in the table from the execution until a definition of the variable is found.

COMMON Variables - Block B		Main Program Variables
B(1):*	B(11):*	N:*
B(2):*	B(12):*	PI:*
B(3):*	B(13):*	I:*
B(4):*	B(14):*	PSF:*
B(5):*	B(15):*	LCRT:*
B(6):*	B(16):*	D1:*
B(7):*	B(17):*	D2:*
B(8):*	B(18):*	P:*
B(9):*	B(19):*	
B(10):*	B(20):*	

*: Value is undefined

AREAR Variables

AREAR:*
A:*
B:*

AREAT Variables

B:*
H:*
AREA:*
AREAT:*

AREAC Variables

AREAC:*
PI:*
RAD:*

DOLS Variables

PSF:*
I:

---

*: Value is undefined

After Execution of Statement #:	Values Defined are:
1	N: I
2	PI: 3.1416
3	I: 1
4	PSF: II LCRT: III D1: IV D2: V
5	
8	
11	
22	B: IV H: V AREA: * (Set from B(1))
23	AREAT: 0.5*IV*V
24	AREAT: *
11	D1: IV D2: V B(1): *
12	
25	PSF: II I: 1
26	B(1): II * (*) (second operand is undefined — it is taken as the value of component I of the second 10-element subarray of COMMON block B. This is element 11 of block B, which is currently undefined (*).)
27	
12	PSF: II I: 1
13	
14	I: 1+1(=2)

Looking at this table, it can be seen, for example, that after execution of statement 23, AREAT's value will be 0.5 times the product of input values IV and V (i.e., the last two values read in by statement 4). Variable I maintains the value 1 from execution of statement 3, until statement 14 is executed. The execution of statement 14 causes the value of I to be incremented by 1. This value is maintained as 2, rather than 1+1, for convenience and comprehensibility. The incremental simplification of formulas as they are evolved is an important feature of symbolic execution. In extensive computations this feature helps to keep the results of the symbolic execution in a simple enough form to enable effective comparison to the intent of the computation.

Some error detection capabilities can be seen from the above example. When statement 13 is executed, the value of B(11) (the alias COST(1) is used by the text) is printed out. It is easy to detect that this value is undefined. Further, the execution of statement 26 shows that B(1) is set by multiplying input value II by an undefined value. These findings suggest the inversion in the COMMON list described earlier. Similarly, the undefinition of AREAT at statement 24, immediately after its definition at statement 23, suggests an error. This discussion makes clear how the symbolic execution of appropriate paths will show that the previously described assertions of functional equality cannot hold true.

Symbolic execution also offers some error detection and analysis capabilities which are qualitatively different from those described above. The specification of an execution path, required as input to symbolic execution, can be used to determine the conditions necessary for execution of the path. This information also has important diagnostic and error detection power.

For example, the path specified above entails a transfer from statement 5 to statement 8. This implies that on execution of statement 5, the value of LCRT.NE.2 must be <u>true</u>. Thus, execution will jump to statement 8 if and only if the current value of LCRT is $\neq 2$. Using the results of symbolic execution, the current symbolic value of every variable is known at all times. In this case, the value III is the current value of LCRT. Thus, the jump from statement 5 to statement 8 will occur if and only if input III is $\neq 2$. The path specification then stipulates a jump from statement 8 to statement 11. This can occur if and only if LCRT.NE.2 evaluates to <u>true</u>. The current symbolic value of LCRT is input value III. Thus, here too, the jump will occur if and only if III $\neq 2$. From this it is inferred that AREAT will be invoked along the path 1, 2, 3, 4,

5, 8, 11 if and only if III, the value read in for LCRT is $\neq$ 2.

Specifically, the values 1 and 3 will cause the execution of AREAT along this path. This deviates from the intent of the program. If that intent were expressed by the assertion

```
C ASSERT I.EQ.3 .AND. I.NE.1.AND.I.NE.2
```

placed after line 11, then symbolic execution could demonstrate the possibility of violating this assertion as well as the conditions under which the violation could occur.

It is intended that statement 6 will be executed if LCRT receives the value 1. This will occur if the path 1, 2, 3, 4, 5, 6 is executed. Using symbolic execution of this path and a constraint examination process, such as shown above, it is determined that this occurs only if the transfer from 5 to 6 is made. This transfer occurs if and only if LCRT.NE.2 evalutes to <u>false</u>. Hence, 1, 2, 3, 4, 5, 6 occurs if and only if III is read in as a 2.

This, too is a deviation from intent. An input value of 2 for LCRT should cause the execution of statement 9 along the path 1, 2, 3, 4, 5, 8, 9. In order for this path to be executed, however, the two symbolic constraints

III $\neq$ 2 (arising from jump (5,8))

and

III = 2 (arising from jump (8,11))

must be simultaneously satisfied. There are no values which can simultaneously satisfy both constraints. Such a set of constraints is inconsistent. If a set of constraints if found to be inconsistent, the path generating those constraints is thereby shown to be unexecutable. Thus, it is shown that statement 9 cannot be reached along the path 1, 2, 3, 4, 5, 8, 9. All of these determinations make clear that there is an error in the control flow of the program. The nature of this error is most clearly indicated as a set of deviations from the intent as captured by flow-of-control specifying assertions.

This example has illustrated the importance of combining constraint solution with symbolic execution. This technique can be used to match statements of computational effect to statements of the conditions under which those effects are obtained. These matchings can be viewed as input-output specifications and can be compared to input-output assertions used to express the intent of the program. This is a powerful verification technique.

Constraint solution can also be coupled with symbolic execution to enable demonstration of the impossibility of violating other classes of assertions.

For example, whenever a division operation is encountered in the process of symbolically executing a path, a new, temporary constraint can be 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, a solution to the system of constraints will produce program input data which forces the traversal of the given input path, followed by a zero-divide error at the given point. If this analytic result can be obtained for all paths leading to a particular division, the need for executing dynamic tests of the divisor is nullified.

In a similar fashion, constraints can be 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). For example, a constraint can be created, constraining the value of N to be ≤ 0 before execution of statement 3 in the program in Figure 1. Using symbolic execution, it is seen that this is not inconsistent with any other constraint, but only constrains input value I to be ≤ 0. Thus, if the first input value to the program is ≤ 0, an illegal DO parameter error will definitely occur when statement 3 is executed. This points out the error of omission previously observed. The variable N should be tested before being used in statement 3.

As a more interesting example, consider the problem of detecting an out-of-bounds array reference error. The previous section indicated that static analysis can be used to verify subscript bounds assertions. This is only true under certain restricted conditions. Suppose, for example, that statement 3 were replaced by

```
3      DO 200 J = 1,N
3.5    I=J
```

or some such similarly obfuscating statement sequence.

Such analysis systems would be hard pressed to properly analyze the full range of possible variations of such code. Hence a more powerful technique such as symbolic execution is needed. Thus assume that the above modification to the source text has been made. Note that such an error will occur at statement 6, for example, if variable I is constrained to be > 10. But symbolic execution along any path leading to statement 6 must go through statement 3.5 and cause the constraint at statement 6 to be expressed as:

(current value of J) > 10

Statement 6 can be reached by path 1, 2, 3, 4, 5, 6 and by all other paths including the statement sequence ..... 12, 13, 14, 4, 5, 6. The only constraint on these sequences are the constraints on the jumps 14 to 4 and 5 to 6. The first constraint can be expressed as:

(current value of J) ≤ (input value I)

The second constraint can be expressed as:

(current value of LCRT) = 2.

Thus, the three constraints are:

(current value of J) > 10

(current value of J) ≤ (input value I)

(current value of LCRT) = 2

This set of constraints simplifies to:

10 < (current value of J) ≤ (input value I)

(current value of LCRT) = 2.

Hence, if input value I is > 10 and input value II = 2, then an array subscript bounds violation error will occur at statement 6.

If the test

```
IF(N.GT.10) STOP
```

is inserted between statements 2 and 3, this will become a constraint incompatible with any out-of-bounds array reference error constraint. Thus, the impossibility of such errors is proven. Systems which automatically generate such error constraints and attempt to solve them current exist [12]. There are theoretical limitations on totally automating them, however.

From the previous discussions, it can be seen that symbolic execution systems are capable of both detecting and disproving possible assertion violations. Error detection or disproof for program computations can be obtained by creating and comparing to assertions the formulas generated by the system. Examination of the systems of constraints arising from the symbolic execution yields other powerful assertion verification capabilities. If a system of constraints leading up to a division is consistent, yet the system augmented by a zero-divisor constraint is inconsistent, then there is no way that a division by zero can occur at that division, provided the division is reached by the given path. Hence, this procedure is capable of demonstrating the absence of division by zero errors along the path. We shall refer to this as pathwise verification. Clearly pathwise verification is not totally satisfactory because an error such as division by zero may still occur in the program simply by

reaching a division along some path other than the one given as input to the symbolic execution system.

The previous discussion also shows that the information gathered about a program by a symbolic execution system is less detailed, but has more general applicability than the information obtained from a dynamic testing system. It is, however, more detailed but less generally applicable than the information obtained from a static analysis 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. This situation is in contrast to the situation for dynamic testing systems, where highly detailed, specific information is obtained, but only relative to a given execution. It also contrasts with the static analysis systems where very vague, but broadly applicable, information is obtained.

Method of Inductive Assertions

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

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

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

The weaknesses of formal verification are its impracticality and unreasonable underlying assumptions. The impracticality is attributable to the formidable amount of work involved in formal verification. The assertion sets required are frequently numerous and their aggregate size is often larger than the code they describe. This gives rise to very large numbers of lemmas and proofs. The proofs of the lemmas are often quite straightforward, but not infrequently they are tricky and intricate. As a result, the entire verification is invariably far larger than the original code, and is shot through with mathematical intricacy. Thus, formal verification requires a sizeable amount of work by a person with a good deal of mathematical facility. This occasions great cost.

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

The interested reader should refer to [16,17,18,19] for further details about this method and for examples of inductive assertion proofs.

Most of these worries about the reliability of formal proofs can be removed by employing an automated verification system such as GYPSY [16]. Such systems rely upon humans primarily for guidance in producing thoroughly detailed, totally justified First Order Predicate Calculus lemmas in support of the formal verification. Despite the size and intricacy of such proofs, a high degree of confidence in their correctness is justified.

More technical proof difficulties cause the imposition of other unreasonable assumptions upon the formal verification process. It is known, for example, that attempted proof of an incorrect lemma may never reduce to an obvious absurdity [19]. Hence, a proof of the incorrectness of an incorrect program cannot necessarily be expected. This is particularly damaging to attempts to use automatic verification systems.

A more serious problem is that formal verification must rely upon an assumed mathematical model of a program's behavior. The purpose of symbolically executing a sequence of code is to determine whether it performs the transformation specified by its bounding assertion sets. The assertion sets are specified in precise mathematical notation. Hence, the effect of executing code must be described mathematically. Many commonly used programming constructs, such as GO TO statements and manipulations on floating point numbers and pointer variables, cannot be readily modeled with complete accuracy. Thus, these techniques, too, have serious limitations in their ability to raise conficence in a program or algorithmic specification.

Finally it is important to stress that even formal verification cannot be used to raise confidence to a level of absolute certainty. Even if adherence to all specified assertions of intent is satisfactorally shown, there may still be reasonable suspicion about whether a complete, accurate set of assertions has actually been specified.

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

V. THE INTEGRATION STRATEGY

In the recent past, each of the four techniques just described (dynamic testing, static analysis, symbolic execution and formal verification) has received considerable attention and investigation. Stand-alone systems, implementing each have been constructed, and experience has been gained in using each. Partly as a result of this experience, there is a growing concensus that no

single technique adequately meets all program testing, verification and analysis needs, but that each contributes some valuable capabilities. It thus becomes clear that the four techniques should not be viewed as competing approaches, but rather that each offers useful but different capabilities. Attention then naturally turns to the examination of how the various capabilities can be merged into a useful total methodology for effectively raising the confidence of diverse users beyond their arbitrarily-set confidence threshholds.

In this section, it shall be shown that the strengths and weaknesses of the four techniques just discussed are fortuitously complementary. It shall be seen that it is reasonable to consider the creation of an integrated methodology for confidence incrementation. The methodology described here makes provision for the progressive detection and exploration of errors leading in a smooth natural way to demonstrations that different aspects of program behavior never violate intent.

As already indicated, the central organizing principle of this methodology is the use of diverse tools and techniques to demonstrate whether differences between actual behavior and asserted intent either exist or are impossible. Central to this methodology is the presence of a specification of intent, in as formal a form as possible. Ideally, intent is expressed in the form of explicit user assertions. At the most basic level, however, a few specifications of intent can be implicitly assumed. For example, it seems safe to assume that a coder intended all divisions to be by non-zero quantities, all references to arrays to be within defined array bounds, and in general that the code produced to be consistent with all rules of the language. Beyond this basic level of intent specification, it is expected that each user will add and then verify explicit assertions of intent incrementally in such a way as to systematically raise his confidence in the code or algorithmic solution at hand.

It is important to observe that the intent specifications, both explicit assertions and basic implicit assumptions, can each be transformed into monitoring probes, as indicated in the section on dynamic testing. As observed there, these probes will indicate the presence of a deviation from intent (error) during a test execution. The absence of such errors during a testing regimen, however, is not sufficient to raise the confidence of most users to a satisfactory level. Thus it is proposed that other techniques be used, where possible, to raise confidence further by demonstrating the absence of error from any possible execution. This is to be done by demonstrating that specific test

probes are redundant or superfluous. Static analysis and formal functional analysis techniques are to be used for this purpose.

Figure 3 summarizes in more detail the flow of the proposed methodology. It shows that confidence raising can be done in a variety of ways, ranging from simple-minded, straightforward and inexpensive to complex, powerful and costly.

In the most straightforward case, one can start only with code or algorithmic specification, and no explicit assertions. Static analysis can be applied, yielding assurances of the absence of such errors as references to uninitialized variables and parameter/argument mismatches. Dynamic testing can be undertaken to yield definitive answers about the absence or presence of array reference and zero division errors from specific test executions. Final and intermediately-computed values can be produced for qualitative assessment by the user. This procedure, widely followed today, is also widely agreed to be insufficient to raise the confidence of most users to satisfactory levels. Hence our proposed methodology indicates stronger measures.

The next stronger step involves the use of more sophisticated static analyzers to examine the instrumented program and attempt to suppress redundant probes prior to dynamic testing. This process should result in a reduction in size of the subject program, without any loss in diagnostic power. Hence the testing process is made more efficient. As observed earlier, the probes remaining are now more likely to indicate the presence of errors. Hence it is desirable to next employ a test path generator (e.g.[20]) to focus testing effort on those situations and specific statements which are most likely to betray errors. In the unlikely case that all probes are removed through static analysis, the sole rationale for doing any dynamic testing would be to obtain a qualitative "feel" for the program by examining test results.

The most significant way in which confidence can be increased is to take the next step, and begin the process of fashioning assertions to express the intent of the code or algorithmic specification. Here again the assertions are to be translated into monitoring code, and imbedded in the program. Direct dynamic testing is now possible, but not recommended. Once again static analysis can and should be used to remove embedded test probes. This will result in a smaller, more efficient test program, and less expensive testing. It will also enable more effective more sharply focused testing, possibly to the point where all probes have been removed and dynamic testing can only yeild a qualitative feel for the execution behavior of the program.

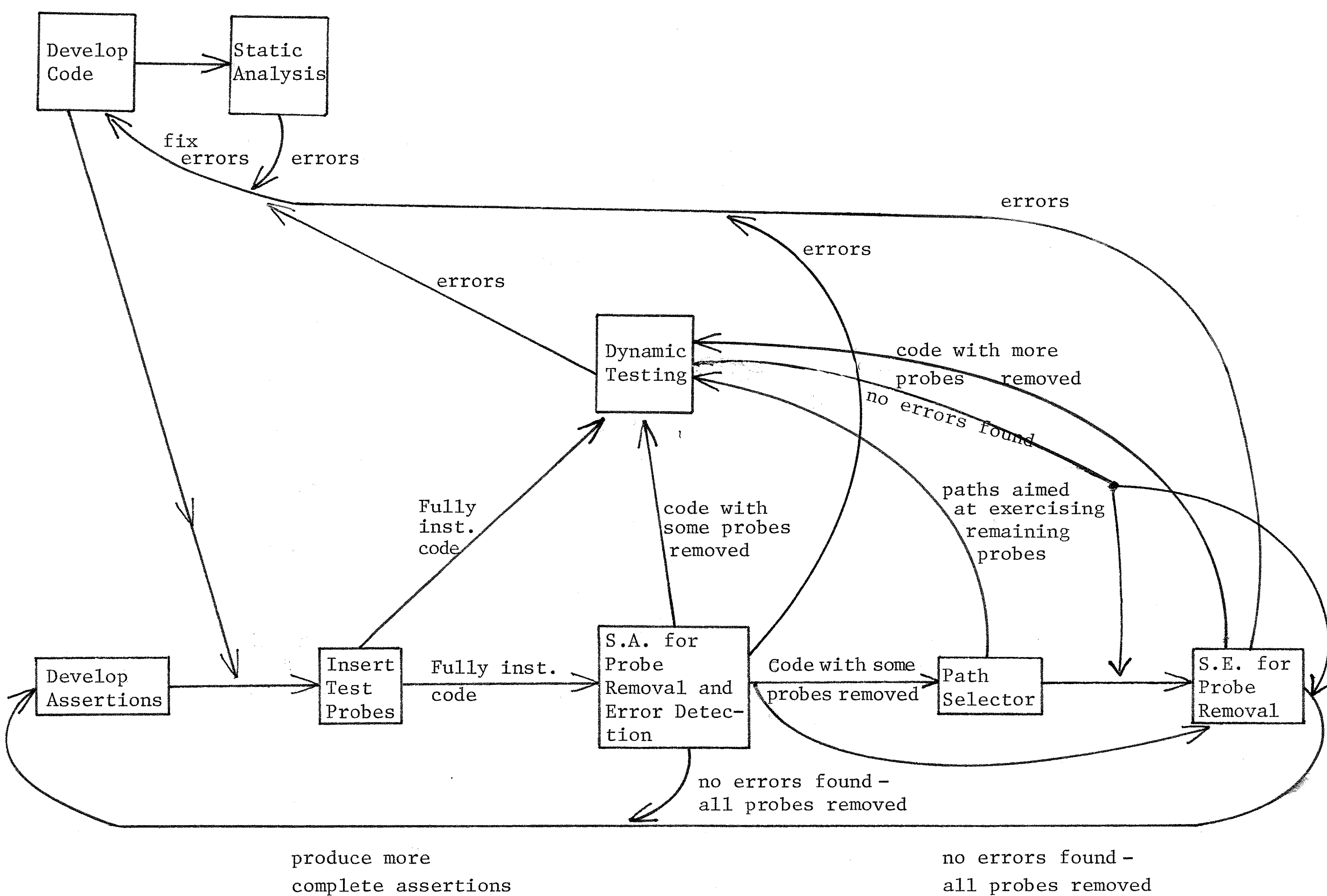

Figure 3

It is more likely that the resulting program will have imbedded probes remaining. In this case, the more sharply focused testing can uncover existing errors, or may fail to do so. If it fails to reveal residual errors, most users would now wish to raise their confidence still further by demonstrating the absence of these errors. This can be done by employing formal functional analysis techniques such as symbolic execution.

As described in an earlier section, symbolic execution can often be used to remove monitoring probes that static analysis could not remove. These may be array bounds or zero division probes, or probes arising out of assertions of functional behavior. This technique is more difficult, expensive and uncertain. Hence it should be attempted cautiously at first, aimed at removal of a small number of selected probes to assure that confidence is being effectively raised at reasonable cost.

After sufficient confidence has been gained (a very subjective decision) the user may wish to create additional assertion sets, capturing more of the range and depth of the program. Incremental confidence gaining is possible now by following the same procedure for testing and removing the probes created by these new assertions. The iterative creation and removal of new assertions should continue as long as the user wishes to gain more confidence and has resources to pursue this procedure.

In the rare case where a user may wish to obtain the strongest formal assurances possible, the user should continue this iterative process until he has expressed the total intent of the program with assertions, and formally removed all probes arising out of these assertions. It is important to caution that, because formal functional analysis techniques make certain incorrect assumptions, as stated earlier, even this process cannot absolutely assure the total absence of errors. Dynamic testing, used in conjunction with judicious test path selection, can and should be used to raise confidence still further.

It can be seen from the preceeding paragraphs that the strategy described organizes the four techniques into a progression of capabilities which is natural in a number of important ways. The strategy begins with a broad scanning procedure and progress to deeper and deeper probing of erros and anomaly phenomena. This initially requires no human interaction or input. It progresses to involve more significant human interaction as human insight becomes more useful in tracing errors to their sources and constructing mathematical demonstrations of correctness. The strategy provides the possibility of some

primitive verification without human intervention, and then allows error detection based upon the negative results of the verification scan.

There also appears to be a natural progression of costs for the processing of the various phases of this strategy. Static analysis appears to be the least expensive, most straightforward operation. It involves a scan over the program text, parse or flowgraph, using encoded information and algorithms which execute rapidly. The symbolic execution phase appears far more costly. It entails performing considerable symbol manipulation and constraint solving for each of the presumably many paths generated by both the static analyzer and path generator. Dynamic testing is likewise relatively costly and, perhaps, more involved than the other two. It entails performing a number of closely monitored executions of the program. Each execution may take as much as several times as long as an unmonitored execution [21] and will generate a large data base of diagnostic information which may have to be probed extensively by human interaction. Formal verification appears to be the most costly of the four techniques. It requires extensive amounts of mathematical labor. It is thus reasonably placed as a high-cost option at the end of the methodology.

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

VI. SOFTWARE LIFECYCLE CONSIDERATIONS

The previous sections of this paper have established the importance of having assertions to represent the intent of a program or algorithmic specification to be tested, analyzed and verified. While the importance of the assertions has been established, the source of the assertions has not been discussed. In this section we propose that the assertions reasonably and naturally originate in the early requirements and design phases of the software production process. We also propose that the testing, verification and analysis techniques already

described are at least partially applicable to these earlier phases.

Figure 4 is a diagrammatic view of how the software production and maintenance process might be divided into phases. It is an adaptation of the "waterfall chart" [22] which has become widely accepted as a model of those activities. The primary goal of these models is to divide software production and maintenance into definable phases and monitoring points. This division should lead to better defined criteria for judging the quality and completeness of work in progress. We shall show how this process also produces assertions and how tools can assist in the process.

The requirements definition phase of this process is the phase during which the basic needs of the software project are enunciated. These needs are to be expressed as precisely and completely as possible, but in such a manner as to not suggest or bias an algorithmic solution. One of the most effective ways to do this is to specify the required functional and performance characteristics of the proposed program. Such a specification need not and should not suggest how the functions are to be computed. These specifications should, from the perspective of this paper, be viewed as assertions of the intent which the eventual program must satisfy. Hence the eventual code assertions must be directly traceable back to these original statements of intent. We shall explore potential mechanisms for doing this shortly.

The preliminary design phase is characterized by the process of exploring possible strategies for building an algorithmic solution which satisfies the requirements specification. During this phase processing modules and data abstractions are defined, and algorithmic processes and data flows are represented, usually hierarchical, showing, when complete, how the principal components of the algorithmic solution are decomposed into successively more detailed specifications of data and processing. In practice, such a decomposition process invariably leads to greater understanding of the problem and consequent changes in requirements. Hence the requirements and preliminary design activities should be viewed as iterative and intertwined. Together they should be considered to be the process of gaining an understanding of the nature of the problem, and agreement about an acceptable approach to its solution.

From the point of view of this paper, preliminary design is important because it specifies the required functional behavior (assertions) which apply to the various components of the solution. Hence this phase begins the process of attaching successively more detailed assertions to successively smaller algorithmic

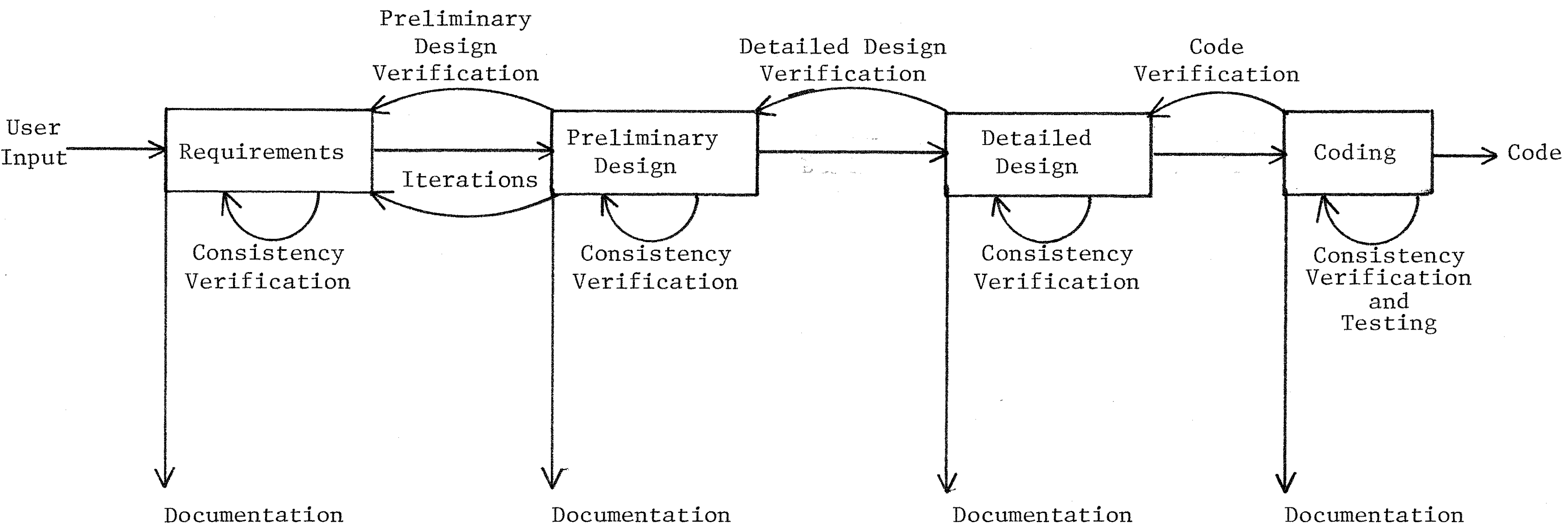

Figure 4

A view of the software production process.

units. This process should terminate with the construction of code around very detailed assertions.

The detailed design phase is the phase during which the outline of the solution, established during preliminary design, is elaborated down to the level of actual specifications for code. Detailed design should not be viewed as merely an extension of the preliminary design activity. At the start of detailed design it is necessary for the designers to reorient their thinking from a problem understanding orientation to a software construction orientation. This is a crucial phase of the software production process, during which the solution elements proposed during preliminary design must be grouped and reorganized into modules and data abstractions [23] [24]. This reorganization should be guided by the desire to clearly capture independent solution concepts in code, and to use standard interfaces to conceal the details of their implementation. The module specifications are statements of the functional behavior required in order to realize the various design concepts. Hence they are assertions. The hierarchical decompositions of the high level modular assertions analogously become assertions specifying the behavior of the submodules comprising higher level modules. The detailed design process terminates with the creation of specifications (assertions) such as those described and shown in earlier sections of this paper, which are so detailed that they can be met with just a few lines of code.

As already noted, one of the primary reasons for following this phased approach to software construction is that it affords obvious opportunities for observing and evaluating progress at intermediate stages. Extensive reviews are conducted at the conclusion of each phase. One of the primary goals of such reviews is to establish whether or not the work completed during that phase meets the objectives as enunciated at the conclusion of the previous phase. Hence the review can quite reasonably be viewed as a testing, analysis and verification procedure, using the output of the previous phase as the statement of intent.

These reviews are invariably based upon documentation and analysis done primarily by humans. It is our contention that they can be heavily supported by tools and techniques like those described earlier in this paper. In order to do this the requirements and design specifications must be stated in terms of a rigorous formalism. Some such formalisms have already been devised. Pseudo-code languages and design representation languages such as CLU [25] are

examples of rigorous formalisms for expressing detailed design. Clearly they can be parsed and subjected to certain types of semantic analysis. Virtually all forms of static analysis and symbolic execution can be carried out on them. Hence some verification and probe removal can be done automatically. If the detailed design and preliminary design are both complete and rigorous enough it is possible to go so far with formal functional analysis as to produce a complete formal verification that the detailed design meets its preliminary design objectives.

It is clearly not possible to dynamically test design specifications in the same way that code can be tested. There is, however, an analog to this process. The specification can be taken to be a model of the way that the completed code is to work, and thus a blueprint for a simulation of that code. A simulation language can be used to create the simulation. Alternatively, the high levels of the design specification can be cast directly into code, and the lowest levels replaced by invocations of stubs, crude simulations of the eventual code. Execution of these simulations can serve many of the same purposes as executing a dynamic test. Clearly the primary difference is an obligatory loss of detail. Good familiarity with specific simulated execution characteristics can, nevertheless be obtained.

It is perhaps more surprising to note that such capabilities can reasonably be extended to requirements and preliminary design specifications. Here again the prerequisite is rigor in the specification. A number of rigorous specification methodologies have been proposed (e.g., SAMM [26], SADT [27], PSL/PSA [28]). All seem to be based upon a graphical representation of the requirements and/or preliminary design.

The SREM methodology [29] is the most interesting as it is strongly supported by the RSL/REVS family of tools [30]. RSL is a language which is used to capture a requirements/preliminary design specification and recast it into a set of objects and relations stored in a centralized data base. The contents of the data base are looked upon as a collection of annotated graphs, modelling the problem and its proposed solution. The REVS system of analytic tools examines the data base and produces documentation, analysis and limited forms of verification. Each processing element in the design has as part of its specification its input/output behavior, and a functional description which may be stated as an algorithmic graph structure. Hence input/output behavior can be statically analyzed and verified for consistency. Symbolic execution

traces can be created as documentation and for the purposes of verification. Dynamic simulations can also be created by REVS to provide an indication of the expected run-time behavior of the eventual coded program. It is important to note that since SREM captures both the requirements and preliminary design in a natural intertwined fashion, verification of internal consistency is tantamount to a verification that preliminary design meets requirements.

We finally are able to see where the code assertions originate. The functional descriptions attached to the various processing elements of an RSL-like specification are the initial program assertions. If the specification technique represents the hierarchical decomposition of these elements, then at each decomposition level functional description is attached to the processing elements. As these descriptions become more algorithmic and rigorous, the possibility of rigorous and automatic verification increases. By the beginning of detailed design they have evolved into rigorous module specifications, and are certainly a suitable basis for the automatic verification approaches described earlier.

Some of the documentation, verification and testing techniques described earlier in connection with code analysis have been applied to requirements and design representations. It remains to be demonstrated that the methodology outlined in Section V and its implementation by the tools proposed can be substantially applied equally well to requirements and design. This would establish the feasibility of a single analytic methodology and tool configuration for application at all phases of the software production process.

VII. REFERENCES

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

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

[3] 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.

[4] M.A. Hennell, D. Hedley, and M.R. Woodward, "Experience with an Algol 68 Numerical Algorithms Testbed," in Proc. Polytech. Inst. of New York Symp. Comput. Software Eng., MRI Symposia Series, vol. XXIV, J. Fox, Ed., Apr. 1976, pp. 457-463.

[5] 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).

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

[7] L.J. Osterweil and L.D. Fosdick, "DAVE--A Validation, Error Detection, and Documentation System for Fortran Programs", Software Practice and Experience 6, pp. 473-486 (Sept. 1976).

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

[9] 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).

[10] 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).

[11] J.D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions", Acta Informatica 2, pp. 191-213 (December 1973).

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

[13] W.E. Howden, "Experiments with a Symbolic Evaluation System", AFIPS 1976 NCC 45, AFIPS Press, Montvale, New Jersey, 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. #75CHO940-7CSR, pp. 234-245.

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

[17] D.I. Good, R.L. London and W.W. Bledose, "An Interactive Program Verification System", Proceedings of 1975 International Conference on Reliable Software, IEEE Cat. #75CHO940-7CSR, pp. 482-492.

[18] R.L. London, "A View of Program Verification", Proceedings of 1975 International Conference on Reliable Software, IEEE Cat. #75CH0940-7CSR, pp. 534-545.

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

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

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

[22] D.J. Reifer, "Automated Aids for Reliable Software," Proc. 1975 International Conference on Reliable Software IEEE Cat. #75-CH0940-7CSR pp. 131-142 (April 1975).

[23] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", CACM 15, pp. 1053-1058 (December 1972).

[24] B.H. Liskov and S.N. Zilles, "Specification Techniques for Data Abstractions", IEEE Trans. on Software Engineering, SE-1, pp. 7-19 (1975)

[25] B.H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU", CACM 20, pp. 564-576, (August 1977).

[26] S.A. Stephens and L.L. Tripp, "A Requirements Expression and Validation Tool", Proceedings Third International Conference on Software Engineering, Atlanta, Georgia (May 1978).

[27] D.T. Ross and K.E. Schoman, Jr., "Structured Analysis for Requirement Definition", IEEE Transactions on Software Engineering, SE-3, pp. 6-15, (January 1977).

[28] D. Teichroew and E.A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, SE-3, pp. 41-48 (January 1977).

[29] M.W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Transactions on Software Engineering, SE-3, pp. 60-69, (January 1977)

This paper presents an approach to integrating four techniques for testing, analysis and verification into one overall strategy for incrementally raising confidence in software in a cost effective way. The paper summarizes the strengths, weaknesses, and operational characteristics of dynamic testing, static analysis, symbolic execution and formal verification. It uses a detailed example as an illustration. Next the integrated strategy is presented. Finally, there is a discussion of how this strategy can be used to raise confidence in software requirements and design specifications as well as in code, thereby making it applicable throughout the entire software lifecycle.

[30] T.E. Bell, D.C. Bixler and M.E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering", IEEE Transactions on Software Engineering, SE-3, pp. 49-60, (January 1977).