TOWARD A THEORY OF TEST DATA SELECTION*

John B. Goodenough
Susan L. Gerhart**
SofTech, Inc., Waltham, Mass.

Keywords and Phrases:

testing, proofs of correctness

Abstract

This paper examines the theoretical and practical role of testing in software development. We prove a fundamental theorem showing that properly structured tests are capable of demonstrating the absence of errors in a program. The theorem's proof hinges on our definition of test reliability and validity, but its practical utility hinges on being able to show when a test is actually reliable. We explain what makes tests unreliable (for example, we show by example why testing all program statements, predicates, or paths is not usually sufficient to insure test reliability), and we outline a possible approach to developing reliable tests. We also show how the analysis required to define reliable tests can help in checking a program's design and specifications as well as in preventing and detecting implementation errors.

1. Introduction

The purpose of this paper is:

- to survey the purpose and limitation of testing as presently conceived,
- to demonstrate a systematic procedure for developing valid and reliable test data,
- to establish a view of the purposes and limitations of testing that will permit testing methods to be systematically improved.

In the remainder of this Introduction, we define some basic concepts relevant to software testing and lay the basis for a theory of test data selection. In Section 2, we discuss two examples of programs published in the literature which contain errors. Our purpose there is to illustrate the problems a theory of testing must deal with. In Section 3, we examine what others have said about the goals, methods, and difficulties of program testing before turning attention to our proposed method, which we present in Section 4. In Section 5, we apply our theoretical concepts to the results of Section 4.

The fundamental questions examined throughout this paper are:

- What are the possible sources of failure in a program?
- What test data should be selected to demonstrate that failures do not arise from these sources?

1.1 Fundamental Testing Concepts

The purpose of testing is to determine whether a program contains any errors. An ideal test, therefore, succeeds only when a program contains no errors. In this paper, one of our goals is to define the characteristics of an ideal test in a way that gives insight into problems of testing. We begin with some basic definitions.

Consider a program F whose input domain is the set of data D. F(d) denotes the result of executing F with input $d \in D$. OUT(d, F(d)) specifies the output requirement for F, i.e., OUT(d, F(d)) is true if and only if F(d) is an acceptable result. We will write OK(d) as an abbreviation for OUT(d, F(d)). Let T be a subset of D. T constitutes an ideal test if OK(t) for all $t \in T$ implies OK(d) for all $d \in D$, i.e., if from successful execution of a sample of the input domain we can conclude the program contains no errors, then the sample constitutes an ideal test. Clearly the validity of this conclusion depends on how "thoroughly" T exercises F. Most papers on testing [e.g., Poole (1973), Stucki (1974)] equate a thorough test with one that is exhaustive, i.e., one for which T = D. But such a definition gives no insight into problems of test data selection. Instead, we define a "thorough" test, T, to be one satisfying COMPLETE(T, C), where COMPLETE is a predicate that defines, in effect, how some data selection criterion, C, is used in selecting a particular set of test data T. C defines what properties of a program must be exercised to constitute a "thorough" test, i.e., one whose successful execution implies no errors in a tested program. COMPLETE insures that T satisfies all these properties. (We will discuss later, in Section 5, the pragmatic reasons for using an incomplete test.)

The data selection criterion C must be defined so tests satisfying COMPLETE (T, C) produce consistent and meaningful results. We express this requirement by saying C must be reliable and valid. In general, reliability refers to the consistency with which results are produced, regardless of whether the results are meaningful. In particular, a data selection

*Supported in part under Contract DAAA25-74-C0469.

**Now at Duke University, Durham, N. C.

Given a program F, with domain D, output requirement OK(d) = OUT(d, F(d)) and test data selection criterion C:

(1) $SUCCESSFUL(T) \equiv (\forall t \in T)\ OK(t)$

(2) $RELIABLE(C) \equiv (\forall T_1, T_2 \subseteq D)(COMPLETE(T_1, C) \wedge COMPLETE(T_2, C) \supset (SUCCESSFUL(T_1) \equiv SUCCESSFUL(T_2)))$

(3) $VALID(C) \equiv (\forall d \in D)\neg OK(d) \supset (\exists T \subseteq D)(COMPLETE(T, C) \wedge \neg SUCCESSFUL(T))$

Fundamental Theorem

$(\exists T \subseteq D)(\exists C)(COMPLETE(T, C) \wedge RELIABLE(C) \wedge VALID(C) \wedge SUCCESSFUL(T)) \supset (\forall d \in D)\ OK(d)$

Figure 1

Formal Definitions and the Fundamental Theorem of Testing

criterion is reliable if and only if every T satisfying COMPLETE(T, C) is processed successfully by F, or if every such T is unsuccessfully processed (see Figure 1). In short to be reliable, C must insure selection of tests that are consistent in their ability to reveal errors, as opposed to necessarily being able to detect all errors. Note that if C is reliable, it is only necessary to test one complete set of test data--no further information will be derived from testing other complete sets of test data.

Validity, in contrast to reliability, customarily refers to the ability to produce meaningful results, regardless of how consistently such results are produced. Hence a test data selection criterion is valid to the extent it does not forbid selection of test data capable of revealing some error. C is valid if and only if for every error in a program, there exists a complete set of test data capable of revealing the error. (See Figure 1 for a formal definition.) But note: validity does not imply that every complete test is necessarily capable of detecting every error in a program. This is the case only if the data selection criterion is reliable as well as valid. Validity only implies it is possible to select data that will reveal an error; it does not guarantee that such data will be selected.

The formal definitions of RELIABLE and VALID given in Figure 1 merely state precisely what we already have said informally about RELIABLE(C) and VALID(C). To show the utility of the definitions, we use them in stating and proving the Fundamental Theorem on which all testing is based (see Figure 1). Its proof is simple:

> Assume there exists some $d \in D$ for which F fails (i.e., $\neg$ OK(d)). Then VALID(C) implies there exists a complete set of test data, T, that is not successful. RELIABLE(C) implies that if one complete test fails, all fail. But this contradicts the theorem's premise, i.e., that there exists a complete test that is successfully executed. Q.E.D.

The theorem is called "basic" not because it is deep--obviously we have constructed our definitions of RELIABLE, VALID and COMPLETE to permit the theorem to be proved. Instead the theorem is basic because of the insight it gives into the concepts of test reliability and validity, and these concepts are basic to the construction of "thorough" tests. The theorem merely demonstrates that tests satisfying COMPLETE(T, C) where C satisfies RELIABLE and VALID are "thorough" in the appropriate sense. Note that proving a data selection criterion to be reliable and valid, and then finding and successfully executing a complete test satisfying this criterion is just a way of proving the correctness of the program. In effect, the theorem states that in some cases, a test is a proof of correctness.

The proof of a selection criterion's validity and reliability is easy in some cases. For example, if C is defined so the only complete test is an exhaustive one, i.e., if $COMPLETE(T, C) \supset (T = D)$, then C obviously satisfies RELIABLE and VALID. Another interesting example is when C is unsatisfiable by any $d \in D$. Then T will be empty, i.e., no testing will be done. In this case, such a C clearly satisfies RELIABLE(C). Proof of C's validity, however, is more difficult. In fact, such a proof exists if and only if the program contains no errors. Hence in this case, proof of C's validity is equivalent to a direct proof of the program's correctness.

A proof of validity is trivial, however, if C does not exclude any member of D from some set of test data, i.e., if it can be shown that for all $d \in D$, there exists a T containing d and satisfying COMPLETE(T, C). In this case, some testing must be performed, and in general, the proof of C's reliability is not trivial. The remainder of the paper will concentrate on what must be known about programs to insure reliability in this case, and in Section 5 we will give some guidelines for finding non-trivial reliable test data selection criteria. Also in Section 5, we will give a specific example of a type of data selection criterion and its corresponding definitions of COMPLETE, RELIABLE, and VALID. But to motivate this example, we first need to look at sources of errors in programs, both in general (Section 1.2) and with reference to example programs (Section 2).

1.2 Types of Program Errors

In general, software errors can be classed as performance errors (failure to produce results within specified or desired time and space limitations) and logic errors (production of incorrect results independent of the time and space required). It is useful to distinguish various types of logic errors:

- construction errors (failure to satisfy a specification through error in an implementation);
- specification errors (failure to write a specification that correctly represents a design);
- design errors (failure to satisfy an understood requirement);
- requirements errors (failure to satisfy the real requirement).

Thorough tests must be able to detect errors arising for any of these reasons, and this means that test data selection criteria must reflect information derived from each stage of software development. Ultimately, however, each type of logic error is manifested as an improper effect produced by an implementation, and for this reason, it is useful to categorize sources of errors in implementation terms:

- Missing Control Flow Paths - This type of error arises from failure to test a particular condition, and hence result in the execution (or non-execution) of inappropriate actions. For example, failure to test for a zero divisor before executing a division may be a missing path error. Other examples will be given later. This type of error results from failing to see that some condition or combination of conditions requires a unique sequence of actions to be handled properly. When a program contains this type of error, it may be possible to execute all control flow paths through the program without detecting the error. This is why exercising all program paths does not constitute a reliable test.

- Inappropriate Path Selection - This type of error occurs when a condition is expressed incorrectly, and therefore, an action is sometimes performed (or omitted) under inappropriate conditions. For example, writing IF A instead of IF A AND B means that when A is true and B false, an inappropriate action will be taken or omitted. When a program contains this type of error, it is quite possible to exercise all statements and all branch conditions without detecting the error. This error can also occur not merely through failure to test the right combination of conditions, but through failure to see that the method of test is not adequate, e. g., testing for the equality of three numbers by writing (X + Y + Z)/3 = X.

- Inappropriate or Missing Action - Examples are calculating a value using a method that does not necessarily give the correct result (e. g., W*W instead of W+W), or failing to assign a value to a variable, or calling a function or procedure with the wrong argument list. Some of these errors are revealed when the action is executed under any circumstances. Requiring all statements in a program to be executed will catch such errors. But sometimes, the action is incorrect only under certain combinations of conditions; in this case, merely exercising the action (or the part of the program where a missing action should appear) will not necessarily reveal the error. For example, this is the case if W*W is written instead of W+W.

This classification of errors is useful because our goal is to detect errors by constructing appropriate tests. Other classifications are useful for other purposes, e. g., to understand the effect of a programming language on software reliability [Rubey (1968)], or to understand why errors occur [Boehm (1975)]. But insight into test reliability is given by the proposed classification. For example, consider the test data selection criterion, "Choose data to exercise all statements and branch conditions in an implementation." In evaluating the reliability of this criterion, we would ask, "Will all construction, specification, design, and requirements errors always be detected by exercising programs with data satisfying this criterion?" Clearly, if a design error, for example, is manifested as a missing path in an implementation, then this criterion for test data selection will not be reliable. So our typology of implementation errors is useful to help understand factors impairing test reliability.

2. Examples of Program Errors

In this section we will discuss errors in two programs that have appeared in the published literature to illustrate the kinds of problems testing must deal with and to lay the groundwork for our proposed testing methodology.

2.1 Example 1: A Simple Text Reformatter

In the article "Programming by Action Clusters," P. Naur (1969) presents the following problem:

> "Given a text consisting of words separated by BLANKS or by NL (new line) characters, convert it to a line-by-line form in accordance with the following rules:
>
> (1) line breaks must be made only where the given text has BLANK or NL;
> (2) each line is filled as far as possible as long as
> (3) no line will contain more than MAXPOS characters."

Naur's purpose is, in part, to justify an approach to algorithm construction "on the basis of General Snapshots needed to prove the algorithm." He does not intend to present a formal proof, but he does present "prescriptions" (assertions) and uses them to guide and justify the construction of action clusters in a top-down manner. So here we have a published article which attempts to use the top-down design method and the guidance of program proving, but the program (see Figure 2) turns out to have at least seven problems.

N1: The program does not terminate when the end of the given text is reached, although Naur provides for termination (through the undefined language construct "Alarm") if a word containing more than MAXPOS characters (an oversize word) is seen. (Note that in this case the specification cannot be satisfied.) Non-termination on end of text will, of course, be discovered with any test data not containing an oversize word. The effect of processing an oversize word, however, would only be seen if test data contained such a word. A reliable test methodology will insure that oversize words are presented to the program (even though they are not mentioned in the specification), since such data are not excluded from the program's input domain and the program will not necessarily process oversize words correctly if it processes shorter words correctly.

```
    bufpos := 0;
    outcharacter(LF);
    fill := 0;

next character:
        incharacter(CW);
        if CW = BLANK ∨ CW = LF
        then begin
            if fill + 1 + bufpos ≤ MAXPOS
            then begin
                outcharacter(BLANK);
                fill := fill + 1  end
            else begin
                outcharacter(LF);
                fill := 0  end;
            for k := 1 step 1 until bufpos do
                outcharacter(buffer [k]);
            fill := fill + bufpos;
            bufpos := 0;   end
        else
            if bufpos = MAXPOS
            then Alarm
            else begin
                bufpos := bufpos + 1;
                buffer[bufpos]  := CW  end;
        go to next character;
```

Figure 2

Naur's original program

```
 1  Alarm := false;
 2  bufpos := 0;
 3  fill := 0;
 4  repeat
 5      incharacter(CW);
 6      if CW = BL ∨ CW = NL ∨ CW = ET
 7      then
 8          if bufpos ≠ 0
 9          then begin
10              if fill + bufpos < MAXPOS ∧ fill ≠ 0
11              then begin
12                  outcharacter(BL);
13                  fill := fill + 1; end
14              else begin
15                  outcharacter(NL);
16                  fill := 0   end;
17              for k := 1 step 1 until bufpos do
18                  outcharacter(buffer [k]);
19              fill := fill + bufpos;
20              bufpos := 0   end
21      else
22          if bufpos = MAXPOS
23          then Alarm := true;
24          else begin
25              bufpos := bufpos + 1;
26              buffer[bufpos]   := CW  end
27  until Alarm ∨ CW = ET;
```

Figure 3

Corrected version of Naur's program

N2: The last word in the text will not be out put unless it is followed by a BLANK or NL.

N3: A blank will appear before the first word on the first line except when the first word is exactly MAXPOS characters long. This can cause a violation of constraint (2) for the first line. The reason for this error is clear. After creating action clusters for a word buffer, Naur makes the following assertion: "The input character preceding the one held in buffer [1] was a BLANK or NL. This has not been output"(p. 252). This assertion is false for the first word and is never disproved. Thus a boundary type of case (first character, first word) causes a proof error. Of course this error will be found for any test data whose first word is less than MAXPOS characters long.

N4: When the first word is MAXPOS characters long, it will be preceded by two line breaks in the output, even though an empty line violates constraint (2).

N5: No provision is made for processing successive adjacent breaks (e. g., two blanks). This error arises because a word is defined as the characters (other than NL or BLANK) appearing between successive NL or BLANK characters, and words of zero length are simply not considered in any discussion of the program's assertions. Specification (2) requires as many words as possible on a line, and this specification makes no sense if zero-length words are permitted. So an important case has not been considered in either the program, the input description, or the program's informal proof. Naur's suggested test data for this program appear to maintain this implicit assumption about the form of the input (i. e., no consecutive BLANKS or NL characters) and do so not reveal the error. How can this sort of error be discovered through a systematic approach to testing?

N6: If the first word of the input text is preceded by a BL or NL, the output will contain either two blanks preceding the first word or a line containing just two blanks, violating constraint (2).

N7: The specifications use NL as the new-line character but the program uses LF. This error probably arises from failure to proof-read, but could also be traced to a failure to specify the character set of the problem. If LF and NL are distinct characters, then any input text containing a NL or LF character will reveal the problem.

What can we conclude from this example?

1. Top-down construction and very informal proofs do not always prevent program errors. Nor does the reading of the description of the program guarantee that errors will be caught. Presumably this paper went through some review process and was proofread, yet the errors persisted. The reviewer in Computing Reviews (Review 19,420) mentioned only the gratuitous blank preceding the first word (error N3). London (1971) corrected errors N1, N3, N4, and N7 but not errors N2, N5, and N6, even though he "proves" his version of the program correct.
2. If this program had been coded and run on some test data, such as that used to

illustrate the program output (p. 251), errors N1, N2, N3, and N7 would have been detected. Errors N4, N5, and N6 would not have been revealed.

Since we wish to use this example to illustrate other types of possible errors and then again in Section 4 to illustrate a general method for selecting test data, we will clean up the specifications and program:

Given an input text having the following properties:

I1: It is a stream of characters, where the characters are classified as break and non-break characters. A break character is a BL (blank), NL (new line indicator), or ET (end-of-text indicator).

I2: The final character in the text is ET.

I3: A word is a nonempty sequence of non-break characters.

I4: A break is a sequence of one or more break characters.

(Thus the input can be viewed as a sequence of words separated by breaks with possibly leading and trailing breaks, and ending with ET.)

The program's output should be the same sequence of words as in the input with the following properties:

O1: A new line should start only between words and at the beginning of the output text, if any;

O2: A break in the input is reduced to a single break character in the output;

O3: As many words as possible should be placed on each line (i.e., between successive NL characters);

O4: No line may contain more than MAXPOS characters (words and BLs);

O5: An oversize word (i.e., a word containing more than MAXPOS characters) should cause an error exit from the program (i.e., a variable Alarm should have the value TRUE);

The corrected version of Naur's program appears in Figure 3. First let's look at the corrections for errors N1 through N7.

N1, N2: The endless loop constructed with a goto in Naur's program has been replaced with a repeat-until having Alarm as a Boolean variable and ET as an end-of-text indicator.

N3: The condition fill $\neq$ 0 has been conjoined with the condition fill + bufpos < MAXPOS to prevent the output of a BL before the first word and to produce a NL instead. The condition fill $\neq$ 0 holds only for the first line, since every other line will contain at least one word. Equally well, we could have initialized fill to the value MAXPOS, insuring that fill + bufpos < MAXPOS would be false the first time.

N4: Line 2 of Naur's program, outcharacter (NL), is removed so that only one NL character will precede the first word of the output text.

N5, N6: An extra predicate, bufpos $\neq$ 0, prevents output when two consecutive breaks occur; the first break forces the word to be output and bufpos to be reset to zero. Note that breaks preceding the first word of the text will also be ignored.

N7: LF was systematically changed to NL throughout the program.

Note how sometimes several error symptoms are corrected with a single program modification; it's often not clear whether the number of errors in a program should be measured by the number of corrections needed to produce a correct program, or by the number of symptoms discovered.

We will now use this program to show how testing based solely on knowledge of a program's internal structure cannot lead to reliable tests. To make our discussion more readily understandable and to lay the groundwork for further discussion in Section 4, we first cast the corrected program into the form of a limited entry decision table [King(1969), Pooch(1974)] (see Table 1). All predicates of the program are written in the condition stub and all assignment and input/output statements in the action stub. The relevant outcomes of the predicates are represented as Y, N, (Y), (N), or --, where (Y) means the outcome is necessarily true, (N) means the outcome is necessarily false, and -- means the outcome can be either true or false. A rule is a single column in the table and specifies a sequence of actions to be performed for a particular condition combination, i.e., a particular set of outcomes of the predicates. For some rules, the outcomes of certain predicates imply that the outcomes of other predicates are either determined (and so of no interest) or are irrelevant. For example, when C4 is true, C6 is necessarily false, and so in rule 1, C6 need not be tested. Predicates whose outcomes are represented as (Y), (N), or -- are not actually evaluated in the program represented by the decision table. For example, the table shows that when C1 is Y, conditions C2 and C6 are not evaluated (see King (1969)).

The fact that the conditions specified in the condition stub are not all independent is represented by the condition constraints given at the right side of the table; it is there, for example, that the relation between the truth of C4 and the falsity of C6 is asserted. The predicates associated with the action stubs (e.g., $(C1 \vee C2) \wedge C3$) indicate under what conditions the associated action is to be performed. Specifying these conditions is not necessary, but provides useful redundancy in checking the correctness of a table.

The decision table format makes it easier to see how various sets of test data satisfy various test data selection criteria. Below the program are written four sets of test data. Each set assumes MAXPOS = 3 and meets some test data selection criterion based on the program's structure. D1 exercises all statements (by exercising rules 3, 5, 9 and 10), D2 all statements and composite predicates, and D3 all statements and individual predicates. *

*An individual predicate is considered to be exercised when it is necessessarily evaluated for some data and it takes on both true and false values. For example, data satisfying rules 1-4 are not considered to exercise C2 because C2 need not be evaluated. C2 is exercised by data satisfying one of rules 5-8 and either rule 9 or rule 10.

Table 1

DECISION TABLE REPRESENTATION OF PROGRAM AND TEST DATA

Initial conditions: ¬C3∧¬C5∧ CW = incharacter (CW) ∧Alarm = FALSE.

		1	2	3	4	5	6	7	8	9	10	
C1:	CW = BL ∨ CW = NL	Y	Y	Y	Y	N	N	N	N	N	N	C1⊃¬C2
C2:	CW = ET	(N)	(N)	(N)	(N)	Y	Y	Y	Y	N	N	C2⊃¬C1
C3:	bufpos ≠ 0	Y	Y	Y	N	Y	Y	Y	N	(Y)	-	¬C3⊃¬C6
C4:	fill + bufpos < MAXPOS	Y	Y	N	-	Y	Y	N	-	(N)	-	C4⊃¬C6; ¬C4⊃ (C3 ∨ C5)
C5:	fill ≠ 0	Y	N	-	-	Y	N	-	-	-	-	¬C3∧¬C5⊃C4; ¬C5∧C6⊃¬C4
C6:	bufpos = MAXPOS	(N)	(N)	-	(N)	(N)	(N)	-	(N)	Y	N	C6⊃C3; C6⊃¬C4
A1 a:	outcharacter (BL)	X				X						(C1∨C2) ∧C3 ∧C4 ∧C5∧¬C6
b:	fill := fill + 1	X				X						
A2 a:	outcharacter (NL)		X	X			X	X				(C1 ∨C2) ∧C3 ∧ (¬C4∨¬C5)
b:	fill := 0		X	X			X	X				
A3 a:	for k := 1 until bufpos outcharacter (buffer [k])	X	X	X		X	X	X				(C1 ∨C2) ∧C3
b:	fill := fill + bufpos	X	X	X		X	X	X				
c:	bufpos := 0	X	X	X		X	X	X				
A4:	Alarm := TRUE									X		¬C1∧¬C2 ∧C6
A5 a:	bufpos := bufpos + 1										X	¬C1∧¬C2∧¬C6
b:	buffer [bufpos] := CW										X	
A6 a:	incharacter (CW)	X	X	X	X						X	C1∨(¬C1∧¬C2∧¬C6)
b:	Repeat table	X	X	X	X						X	
A7:	Exit table					X	X	X	X	X		C2 ∨(¬C1 ∧C6)
	Test Data	Rule Exercised										Rule Sequence Exercised
D1.1	A, A, A, A, ET									X	X	10, 10, 10, 9
D1.2	A, A, A, BL, B, BL, C, ET			X		X					X	10, 10, 10, 3, 10, 3, 10, 5
D2.1	A, A, A, A, ET									X	X	10, 10, 10, 9
D2.2	A, A, A, BL, B, BL, C, NL, ET	X_{NL}		X_{BL}					X		X	10, 10, 10, 3, 10, 3, 10, 1, 8
D3.1	A, A, A, A, ET									X	X	10, 10, 10, 9
D3.2	A, BL, B, NL, C, NL, ET	X_{BL}	X_{NL}	X_{NL}					X		X	10, 2, 10, 1, 10, 3, 8
D4.1	A, A, A, A, ET									X	X	10, 10, 10, 9
D4.2	A, BL, BL, B, BL, C, NL, ET	X_{BL}	X_{BL}	X_{NL}	X_{FL}				X		X	10, 2, 4, 10, 1, 10, 3, 8
D4.3	A, ET						X				X	10, 6
D4.4	A, BL, B, ET		X			X					X	10, 2, 10, 5
D4.5	A.BL, B, NL, C, BL, D, D, D, ET	X_{NL}	X_{BL}	X_{BL}				X			X	10, 2, 10, 1, 10, 3, 10, 10, 10, 7

D3 also exercises all loop iteration paths, i.e., all paths through a loop that are possible on some iteration of the loop; by exercising all individual predicates as well, D3 ensures all loop exiting conditions are also tested. D4 exercises all rules of the table. Note that exercising all rules is equivalent to exercising each statement under all condition combinations considered relevant by the writer of the program. Table 2 shows more clearly what combination of rules need to be exercised to satisfy the various test data selection criteria. as well as how the data actually satisfy the criteria. For example, this table shows that the composite predicate C4∧C5 (i.e., fill + bufpos < MAXPOS ∧ fill ≠0) is true for any data exercising rules 1 and 5 and false for any data exercising rules 2, 3, 6 or 7. For data exercising any other rule, the value of this composite predicate is irrelevant. The actual test data exercise rules 1 and 3.

We wish to show that none of these exercising criteria are completely reliable, i.e., it is possible to exercise a program containing an error using any of these criteria without necessarily discovering the error. This shows that tests based solely on a program's internal structure are unreliable; their success is poor evidence that a program contains no errors.

Five examples of errors are summarized in Table 3. The errors can be characterized as follows:

1) an incorrect predicate (fill + bufpos ≤MAXPOS instead of fill + bufpos <MAXPOS) causing rules 1 or 5 to be selected under circumstances where rules 3 or 7 should have been selected. This is an inappropriate path selection type of construction error. It will not be detected by any of the four test data sets. To

Table 2
RULES EXERCISED TO SATISFY VARIOUS COMPLETENESS CRITERIA

1. Composite Predicates

Composite Predicates	Rules To Be Exercised		Rules Exercised By D2		Rules Exercised By D1	
	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
C1 ∨ C2	(1-8)	(9, 10)	1, 3	9, 10	3	9, 10
Alarm ∨ C2	(5-9)	(1-4, 10)	8, 9	1, 3, 10	5, 9	10
C3	(1-3, 5-7)	(4, 8)	1, 3	8	3, 5	
C4 ∧ C5	(1, 5)	(2, 3, 6, 7)	1	3	5	3
C6	(9)	(10)	9	10	9	10

2. Individual Predicates

Individual Predicates	Rules To Be Exercised		Rules Exercised By D3		Rules Exercised By D2	
	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
$C1_{BL}$	(1-4)	(1-10)	1, 2	3, 8, 9, 10	1	3, 8, 9, 10
$C1_{NL}$	(1-4)	(5-10)	3	8, 9, 10	3	8, 9, 10
C2	(5-8)	(9, 10)	8	9, 10	8	9, 10
C3	(1-3, 5-7)	(4, 8)	1, 2, 3	8	1, 3	8
C4	(1, 2, 5, 6)	(3, 7)	1, 2	3	1	3
C5	(1, 5)	(2, 6)	1	2	1	
C6	(9)	(10)	9	10	9	10
Alarm	(9)	(10)	9	10	9	10

3. Loop Iteration Paths

Distinct Paths	Rules Exercised By D3
(1, 5)	1
(2, 3, 6, 7)	2, 3
(4, 8)	8
(9)	9
(10)	10

Table 3
EFFECT OF ERRORS

Error in Program	Error Type	Effect on Table
1. Change fill + bufpos < MAXPOS to fill + bufpos ≤ MAXPOS	inappropriate path selection	Condition C4 changed similarly.
2. Omit line 13 fill:= fill + 1	missing action	No mark on Action Alb.
3. Omit bufpos ≠ 0 test (line 8) (This is Naur's error N5, 6.)	missing path	Condition C3 line will be removed and therefore rules 4 and 8 can be dropped.
4. Omit fill ≠ 0 from line 10 (This is Naur's error N3.)	inappropriate path selection	Condition C5 removed from table. Rules 2 and 6 are eliminated, being subsumed under rules 1 and 5.
5. Omit CW = ET from line 6	inappropriate path selection	Rules 5-8 eliminated and rule 9 is modified. See Table 4.

Table 4

DECISION TABLE REPRESENTATION OF PROGRAM CONTAINING ERROR 5

		1	2	3	4	5	6	7	8'	9'	10	
C1:	CW = BL ∨ CW = NL	Y	Y	Y	Y				N	N	N	C1⊃¬C2
C2:	CW = ET	(N)	)N)	(N)	(N)				Y	-	N	C2⊃¬C1
C3:	bufpos ≠ 0	Y	Y	Y	N				-	(Y)	-	¬C3⊃¬C6
C4:	fill + bufpos < MAXPOS	Y	Y	N	-				-	(N)	-	C4⊃¬C6; ¬C4⊃ (C3 ∨ C5)
C5:	fill ≠ 0	Y	N	-	-				-	-	-	¬C3∧¬C5⊃C4; ¬C5 ∧ C6⊃¬C4
C6:	bufpos = MAXPOS	(N)	(N)	-	(N)				N	Y	N	C6⊃C3; C6⊃¬C4
A1 a:	outcharacter (BL)	X										C1∧ C3 ∧ C4 ∧C5∧¬C6
b:	fill := fill + 1	X										
A2 a:	outcharacter (NL)		X	X								C1 ∧C3 ∧ (¬C4∨¬C5)
b:	fill := 0		X	X								
A3 a:	for k := 1 until bufpos outcharacter (buffer [k])	X	X	X								C1 ∧C3
b:	fill := fill + bufpos	X	X	X								
c:	bufpos := 0	X	X	X								
A4:	Alarm := TRUE									X		¬C1∧ C6
A5 a:	bufpos := bufpos + 1								X		X	¬C1∧¬C6
b:	buffer [bufpos] := CW								X		X	
A6 a:	incharacter (CW)	X	X	X	X						X	C1 ∨ (¬C6∧¬C2)
b:	Repeat table	X	X	X	X						X	
A7:	Exit table								X	X		C2 ∨ (¬C1 ∧C6)

show this error, the loop must be executed with fill + bufpos = MAXPOS and fill ≠ 0. This requires at least MAXPOS + 2 iterations of the loop and the word lengths must be just right. Although this is an "off-by-one" type of error, not just any data such that fill+bufpos = MAXPOS will reveal it, e.g., D1.2 and D2.2 do not do so.

2) a missing action (fill := fill + 1) yielding essentially the same effect as error 1 when there are only two words on a line. This error also will not be detected by any of the four test data sets.

3) a missing predicate (bufpos ≠ 0) yielding a missing path type of error that would not be detected with the D1 data set. (This is one of Naur's errors.) Although the other data sets would detect this error, different data sets can be constructed that will not detect the error and yet will satisfy the various exercising criteria for the erroneous program. For example, D1 as it stands would exercise all statements and composite predicates in the modified program (see Table 2 with C3 eliminated). Also, test D4.2 could then be eliminated and the other D4 data would then be sufficient to exercise all rules, interior loop paths, and individual predicates without revealing the error.

4) a missing predicate (fill ≠ 0) causing an inappropriate path selection. (This is also one of Naur's errors.) Alternatively, this error could be characterized as an inappropriate action (namely, fill := 0 in line 3 should be replaced with fill := MAXPOS). This error is easily detected by any data whose first word is less than MAXPOS characters long. Note that D1 and D2 do not detect this error precisely because the first word is exactly MAXPOS characters long. In fact, data set D2 exercises all individual predicates, interior loop paths, and statements in the erroneous program without showing this error. It is also readily possible to devise data to exercise all rules in a decision table representation of the erroneous program without revealing the error.

5) a missing predicate (CW = ET in line 6) causing inappropriate path selection. This is essentially error N2. This error will not be found by D2 or D3. It also is not necessarily found by data exercising all rules in the decision table representing the altered program (see for example, Table 4; data D4.1 and D4.2 exercise all the rules in this table without revealing an error). This error shows that even when all conditions relevant to the correct operation of a program appear explicitly in the program, test data selection criteria based solely on a program's internal structure will not necessarily guarantee tests that will reveal the error.

This analysis of five possible errors shows the unreliability of test data selected merely to exercise all statements, composite predicates, individual predicates, loop iteration paths, or rules in a program's decision table representation. These examples of errors show that:

Table 5
ALL FEASIBLE RULES IMPLIED BY TABLE 1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
C1: CW = BL ∨ CW = NL	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	C1⊃¬C2
C2: CW = ET	(N)	(N)	(N)	(N)	(N)	(N)	(N)	(N)	(N)	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	C2⊃¬C1
C3: bufpos ≠ 0	Y	Y	(Y)	(Y)	Y	Y	N	N	N	Y	Y	(Y)	(Y)	Y	Y	N	N	N	Y	Y	(Y)	(Y)	Y	Y	N	N	N	¬C3⊃¬C6
C4: fill + bufpos < MAXPOS	Y	Y	(N)	(N)	N	N	Y	N	(Y)	Y	Y	(N)	(N)	N	N	Y	N	(Y)	Y	Y	(N)	(N)	N	N	Y	N	(Y)	C4⊃¬C6; ¬C4⊃ (C3 ∨ C5)
C5: fill ≠ 0	Y	Y	Y	N	N	Y	(Y)	(Y)	N	Y	N	Y	N	N	Y	(Y)	(Y)	N	Y	N	Y	N	N	Y	(Y)	(Y)	N	¬C3∧¬C5⊃C4; ¬C5∧C6⊃¬C4
C6: bufpos = MAXPOS	(N)	(N)	Y	Y	N	N	(N)	(N)	(N)	(N)	(N)	Y	Y	N	N	(N)	(N)	(N)	(N)	(N)	Y	Y	N	N	(N)	(N)	(N)	C6⊃C3; C6⊃¬C4
Table 1 Rule Numbers	1	2	3	3	3	3	4	4	4	5	6	7	7	7	7	8	8	8	10	10	9	9	10	10	10	10	10	
A1 a: outcharacter (BL)	X									X																		(C1 ∨ C2) ∧ C3 ∧ C4 ∧ C5 ∧ ¬C6
b: fill := fill + 1	X									X																		
A2 a: outcharacter (NL)		X	X	X	X	X					X	X	X	X	X													(C1 ∨ C2) ∧ C3 ∧ (¬C4 ∨ ¬C5)
b: fill := 0		X	X	X	X	X					X	X	X	X	X													
A3 a: for k := 1 until bufpos outcharacter (buffer [k])	X	X	X	X	X	X				X	X	X	X	X	X													(C1 ∨ C2) ∧ C3
b: fill := fill + bufpos	X	X	X	X	X	X				X	X	X	X	X	X													
c: bufpos := 0	X	X	X	X	X	X				X	X	X	X	X	X													
A4: Alarm := TRUE																					X	X						¬C1∧¬C2∧C6
A5 a: bufpos := bufpos + 1																			X	X			X	X	X	X	X	¬C1∧¬C2∧¬C6
b: buffer [bufpos] := CW																			X	X			X	X	X	X	X	
A6 a: incharacter (CW)	X	X	X	X	X	X	X	X	X										X	X			X	X	X	X	X	C1∨(¬C1∧¬C2∧¬C6)
b: Repeat table	X	X	X	X	X	X	X	X	X										X	X			X	X	X	X	X	
A7: Exit table										X	X	X	X	X	X	X	X	X			X	X						C2 ∨ (¬C1 ∧ C6)
Test Data	Rules Exercised																											Rule Sequence Exercised
DX.1 A, A, A, A, ET																			X		X						X	27, 20, 20, 22
D1.2 A, A A, BL, B, BL, C, ET					X	X				X									X						X		X	27, 20, 20, 5, 25, 6, 25, 10
D2.2 A, A, A, BL, B, BL, C, NL, ET	X				X	X											X		X						X		X	27, 20, 20, 5, 25, 6, 25, 1, 17
D3.2 A, BL, B, NL, C, NL, ET	X	X				X										X									X	X	X	27, 2, 25, 1, 26, 6, 16
D4.2 A, BL, BL, B, BL, C, NL, ET	X	X				X	X									X									X	X	X	27, 2, 7, 25, 1, 26, 6, 16
D4.3 A, ET											X																X	27, 11
D4.4 A, BL, B, ET		X								X															X		X	27, 2, 25, 10
D4.5 A, BL, B, NL, C, BL, D, D, D, ET	X	X				X						X							X					X	X	X	X	27, 2, 25, 1, 26, 6, 25, 19, 24, 12

1) To detect errors reliably, it is in general necessary to execute a statement under more than one combination of conditions to verify that its effect is appropriate under all circumstances. Exercising any statement just once is usually inadequate.

2) Equally well, the same path through a loop will usually have to be exercised more than once before the right combination of conditions is found to reveal a missing path or inappropriate path selection error. For example, error 5 (see Table 3) requires exercising rule 8' of Table 4 with bufpos ≠ 0 to show the error. This path should also be exercised with bufpos = 0 to guard against some other error, e.g., error 3, even though bufpos's value is not even tested when executing rule 8'.

In short, a reliable test is designed not so much to exercise program paths as to exercise paths under circumstances such that an error is detectable if one exists. Tests based solely on the internal structure of a program are likely to be unreliable.

To find data that reliably reveal errors like those illustrated here, all the conditions relevant to the correct operation of a program must be known and test data must be devised to exercise all possible combinations of these conditions, whether or not the program's internal structure indicates certain condition combinations cause different sequences of actions to be performed. For example, Tables 1 and 4 actually represent 27 distinct rules if all "don't care" conditions are written out explicitly as different rules (see Table 5). Test data required to exercise all 27 rules will detect error 5 because this error in effect combines rules 12, 13, 21, and 22 to form rule 9' and rules 10, 11, 14-18 to form rule 8'. If for errors 3 and 4, conditions C5 and C3 are retained in Table 5 even though they are not tested in the program, test data to detect these errors will necessarily be generated if all 27 condition combinations are exercised. Errors 1 and 2 will be found if the condition fill + bufpos = MAXPOS is added to the table. Data to exercise this condition in combination with all the others will necessarily reveal errors 1 and 2. (This condition adds only six rules to Table 5 because it is not independent of the other conditions.)

In short, the secret of reliable testing is to find all conditions relevant to a program's correct operation and to exercise all possible combinations of these conditions. In Section 4, we will illustrate how relevant conditions can be discovered.

2.2 Example 2: An Exam Scheduler

This program is presented by Hoare in Dahl (1972) as an example of the use of various abstract data structures and an informal proof of correctness.

The problem is to construct a timetable for university examinations such that

1) The number of sessions is kept small, though no absolute minimum is stipulated.
2) Each exam is scheduled for one session.
3) No exam is scheduled for more than one session.
4) No session involves more than K exams.
5) No session involves more than h students.
6) No student takes more than one exam in a session.

The solution proposed by Hoare tries all combinations of unscheduled exams that satisfy (2) through (6) and selects the combination that maximizes h.

The specifications and assertions, except (1) are precisely and formally stated such that a formal proof could be performed. However, an error creeps in related to (1). It is impossible to state (1) precisely since there is no way to predetermine the number of sessions, so it is assumed that the solution method will do the best it can. The error arises from a variable "remaining" which is to contain all those exams which do not appear in the table so far, i.e., unscheduled exams. The error is manifested in two ways:

1) Performance. Exactly 500 exams are always processed even if there are fewer than 500 exams to be scheduled. Since the program is recursive and searches all combinations of unscheduled exams, the performance will be degraded by processing "empty exams" (exams for which no student is registered).

2) Violation of constraints. Empty exams still have to be scheduled in some session, so it is possible that the limitation implicit in (1) could be violated unnecessarily. That is, the program as it stands may be unable to produce a solution where it should if empty exams were not processed.

This error is easy to correct by initializing "remaining" to include only those exams for which at least one student has registered.

This error is very different from those of Example 1 and is important for the following reasons:

1) This program would be very hard to understand and to test thoroughly without some sort of an informal proof to guarantee that all combinations are properly handled. But the proof only considers assertions (2) - (6) and ignores (1) because it can't be formalized. The error then arises because (1) is not checked out informally even if it can't be formally stated and proved. Even formally specifiable necessary conditions for (1) to be satisfied are not proved, e.g., that no session schedules an empty exam. This error shows the danger of concentrating on the difficult parts of a program and the formalized specifications to the exclusion of the easier parts.

2) This program is similar to many other programs where the correct answer can't be formally characterized and where it would be difficult to even determine whether the produced timetable has the fewest number of sessions.

Assuming no proof existed or as confirmation of the proof, what type of test data should be selected? What criteria for test data can hope to yield a reliable test of this program? Our proposed approach is not yet sufficiently powerful to answer these questions. We cite this example to show the difficulties that must be resolved by a complete theory of test data selection as well as to show that these difficulties challenge even a proof of correctness approach to software reliability.

3. Views on Testing

Having discussed some fundamental definitions in the testing area and examined examples of the problems to be solved by an approach to testing, we will now look at what others have said about the goals, methods, and difficulties of program testing before turning our attention to our proposed method.

3.1 "Exhaustive" testing

> "Software certification... ideally... means checking all possible logical paths through a program." [Boehm (1973)]

> "Exhaustive testing, i.e., testing all possible (input)..." [Poole (1973), p. 310]

Exhaustive testing, defined either in terms of program paths or a program's input domain (although these definitions are not equivalent)*, is usually cited as the impractical means of achieving a reliable and valid test. The importance of the cited statements is in the questions which arise from them:

- Is there a way to test a program which is equivalent to exhaustive testing (in the sense of being reliable and valid)?
- Is there a practical approximation to exhaustive testing?
- If so, how good is the approximation?

It should be noted that exhaustive testing of a program's input domain is a process not necessarily guaranteed to terminate. There are some programs whose behavior (e.g., whether they stop) is impossible to verify by testing or any other means and some programs have infinite input domains, so exhaustive testing can never be completed.

Although some programs cannot be exhaustively tested, a basic hypothesis for the reliability and validity of testing is that the input domain of a program can be partitioned into a finite number of equivalence classes such that a test of a representative of each class will, by induction, test the entire class, and hence, the equivalent of exhaustive testing of the input domain can be performed. If such tests all terminate and if the partitioning is appropriate, a completely reliable test of the program will have been performed. This is not, of course, a novel idea. Hoare [in Buxton (1970), p. 21] has pointed out that the essence of testing is to establish the base proposition of an inductive proof (we pursue this idea further in Section 5). This pinpoints the fundamental problem of testing--the inference from the success of one set of test data that others will also succeed, and that the success of one

---

*Exhaustive testing of paths will not necessarily find all errors. For example, both paths in the following program can be tested without necessarily revealing the error:

```
IF (X+Y+Z)/3 = X
    THEN PRINT ("X, Y, and Z are equal in value");
    ELSE PRINT ("X, Y, and Z are not equal in
                                        value");
```

test data set is equivalent to successfully completing an exhaustive test of a program's input domain.

3.2 Is Proving Really Better Than Testing?

> "Program testing can be used to show the presence of bugs, but never to show their absence!" [in Dahl (1972), p. 6]

> "[When] you have given the proof of [a program's] correctness, ... [you] can dispense with testing altogether." [in Naur (1969b), p. 51]

Just because testing is not a completely reliable means of demonstrating program correctness does not mean it's sensible to rely solely on proofs. Proofs aren't completely reliable either. Proofs can only provide assurance of correctness if all the following are true:

a. There is a complete axiomatization of the entire running environment of the program--all language, operating system, and hardware processors.

b. The processors are proved consistent with the axiomatization.

c. The program is completely and formally implemented in such a way a proof can be performed or checked mechanically.

d. The specifications are correct in that if every program in the system is correct with respect to its specifications, then the entire system performs as desired.

These requirements are far beyond the state of the art of program specification and mechanical theorem proving, and we must be satisfied in practice with informal specifications, axiomatizations, and proofs. Then problems arise when proofs have errors, specifications are incomplete, ambiguous, or unformalizable (as in example 2), and systems are not axiomatizable. The two examples already discussed have clearly shown that an incomplete attempt at a program proof does not assure a program will not fail. These examples are very realistic in terms of the state of the art of program proving for real programs.

Despite the practical fallibility of proving, attempts at proof are nonetheless valuable. They can reveal errors and assist in their prevention. Furthermore, programs cannot be proved unless they can be understood. Facilitating provability sets a worthy standard for good languages, program structure, specifications, and documentation, and thereby assists in preventing errors.

In short, neither proofs nor tests can, in practice, provide complete assurance that programs will not fail. Even so, attempts to prove programs, like attempts to test them completely, are essential to the development of usable programs. Tests have the advantage of providing accurate information about a program's actual behavior in its actual environment; a proof is limited to conclusions about behavior in a postulated environment. Both have an essential role in program development, though each is a fallible technique. Testing and proving are complementary methods for decreasing the likelihood of program failure.

What testing lacks (and what this paper is attempting to provide a start towards) is a theoretically sound (but practical) definition of what constitutes an adequate test. When our understanding of what makes an adequate test is as good as our understanding of what makes an adequate proof, testing can assume its proper role in the theory and practice of software development.

3.3 The Effect of Testing Considerations on Program Development

> "Testing must be planned for throughout the entire design and coding process" [Hetzel (1973), p. 18]

Testing must not only be planned for, it is in fact performed in one way or another at every stage of the programming process, not just when a program has been completed. Specifications must be tested against examples to see if they are complete, consistent, and unambiguous; programs are desk checked by informally considering what will happen when certain kinds of data are processed; a proof must partition data into cases that affect the validity of assertions differently. Testing, in the sense of identifying distinguishable cases and evaluating their effect on the programming requirement, specifications, code, or a proof, occurs in every phase of programming. Moreover, the need to test affects the process of program construction in various ways:

a) Testing being inevitable, it is good practice to identify testing needs, e.g., weak or critical links in a system, early in program design (e.g., see Hansen (1973)).

b) Programs should be structured so logical testing of various abstractions of the program can reduce actual testing of the final program.

c) Specifications must be precise enough to be testable.

d) The need to get special information just to verify the successful execution of a test run affects program design, (e.g., testing an operating system scheduler requires access to information not ordinarily available).

Improved understanding of the nature of testing will help in developing and performing tests at all stages in the program development process.

3.4 Conclusions

There is a general concept which unifies all the activities involved in demonstrating that a program does not fail. That concept is case analysis. Case analysis occurs in

a) Design and specification where it is necessary to exclude cases of data where the program is not expected to operate or to identify cases in the input or output which require special treatment. In Example 1, these would include successive break characters and termination characters, and in Example 2 the empty exam.

b) Program construction, where the solution to the problem dictates that certain cases require

special treatment and other cases can be lumped together. These are illustrated in Example 1 where the predicates bufpos ≠ 0 and fill ≠ 0 form paths which treat successive break characters and the first word on the first line, respectively, and in Example 2 where the empty exam must be filtered out of the exams to be processed.

c) Program proving, where assertions must be made about special cases or generalized to cover several cases and where proofs break into cases. Example 1 shows an assertion which must have a case which covers the first word as well as the case that covers all non-first words and Example 2 shows that the assertion which is not formalized must be checked out in a different manner from the more formalized assertions.

d) Testing, where it must be demonstrated that the total of all cases involved in program specification, design, construction, and proof have been correctly and completely handled. Ideally, the cases created during design specification, construction, and proof should be an adequate basis for test data selection, although the cases may need further analysis and combination in order to obtain a reasonably sized and still effective set of test data.

There is a great need for more examples to make vague testing concepts (e.g., "boundary" condition) at least more intuitively clear and for theories of test data selection which try to analyze the facts about testing in relation to each other for the purpose of guiding the selection of test data toward greater effectiveness. Section 4 will illustrate a technique for selecting test data and Section 5 will analyze this technique in terms of the theory sketched in Section 1.

4. Developing Effective Program Tests

Our goal is to define how to select test data for a particular program such that if the program processes all test data correctly, we can be reasonably confident the program will process all data correctly. For this to be possible, the test data must cover all errors in the program being tested, i.e., test data selection criteria must ensure that data revealing any errors will be selected.

Our approach to testing (and indeed, any approach) is based on assumptions about how program errors occur. Our approach assumes errors are primarily due to 1) inadequate understanding of all conditions a program must deal with, and 2) failure to realize that certain combinations of conditions require special treatment. We distinguish:

- Test data, the actual values from a program's input domain that collectively satisfy some test data selection criterion;
- Test predicates, a description of conditions and combinations of conditions relevant to the program's correct operation.

In short, test predicates describe what aspects of a program are to be tested; test data cause these aspects to be tested.

Our purpose in this part of the paper is 1) to illustrate the problems and issues faced in developing reliable and valid tests for a particular program so that later, in Section 5, when we discuss formal criteria for reliability and validity, examples illustrating the formal criteria will be at hand; and 2) to show informally, what may, with further research, become a practical approach to defining reliable tests.

4.1 An Overview of the Method

There are several sources of information about the conditions and combinations of conditions relevant to a program's operation:

- the general requirement a program is to satisfy;
- the program's specification;
- general characteristics of the implementation method used, including special conditions relevant to data structures and how data is represented; and
- the specific internal structure of an actual implementation.

None of these sources of information is by itself sufficient for generating a reliable test. Information from all these sources must be used to insure all combinations of conditions are identified.

Our technique for developing and describing test predicates is derived from decision table techniques, and will be called the condition table method. A condition table looks like the top half of a decision table. Each column in a condition table is a logically possible combination of conditions that can occur when executing the program being described. Each column is our representation of a test predicate. The basic idea is to identify conditions describing some aspect of the problem or program to be tested, and then build condition tables for groups of conditions. Sometimes separate condition tables are combined into a single table; when to combine tables and when to leave them separate is an important issue needing further study.

4.2 Deriving Test Cases from Specifications

A program's specifications are an important source of test because data satisfying such predicates are able to detect missing path and inappropriate path selection construction errors. Such data can also detect design and specification errors, as we shall see. We will use Naur's specifications for Example 1 to illustrate test case development from specifications. Then, using knowledge of an actual implementation (either that shown in Figure 2 or Figure 3), we will show how to eliminate some test predicates without impairing test reliability. (It may also happen that some test predicates will have to be added when information about an actual implementation becomes available, but this does not occur here.)

The first part of Naur's specification states:

(1) "Given a text, consisting of words separated by BLANKs or NL (new line) characters, ..."

This clause attempts to describe the program's input domain. We begin to develop a condition

Table 6

FIRST CONDITION TABLE FOR INPUT DOMAIN

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C1: PrevChar[BL, NL, Ot]	BL	BL	BL	BL	NL	NL	NL	NL	Ot	Ot	Ot	Ot	?	?	?	?
C2: CurChar[BL, NL, Ot]	BL	NL	Ot	?	BL	NL	Ot	?	BL	NL	Ot	?	BL	NL	Ot	?

Table 7

REVISED CONDITION TABLE FOR INPUT DOMAIN

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
C1: PrevChar[BL, NL, Ot]	BL	BL	BL	BL	BL	BL	BL	BL	BL	BL	BL	BL	NL	NL	NL	NL	NL	NL	NL	NL	NL	NL	NL	NL	Ot	Ot	Ot	Ot	?	?	?	?
C2: CurChar[BL, NL, Ot, ET]	BL	BL	NL	NL	Ot	Ot	Ot	Ot	ET	ET	ET	ET	BL	BL	NL	NL	Ot	Ot	Ot	Ot	ET	ET	ET	ET	BL	NL	Ot	ET	BL	NL	Ot	ET
C3: First Word?	Y	N	Y	N	Y	Y	N	N	Y	Y	N	N	Y	N	Y	N	Y	Y	N	N	Y	Y	N	N	(Y)	(Y)	(Y)	(Y)	(N)	(N)	(N)	(N)
C4: Break length[1, >1]	?	?	?	?	1	>1	1	>1	1	>1	1	>1	?	?	?	?	1	>1	1	>1	1	>1	1	>1	?	?	?	?	?	?	?	?

Constraints between conditions

C2(BL) ∨ C2(NL) ⊃ C4(?)
C1(Ot) ⊃ C3
C1(?) ⊃ ¬C3

table by looking for conditions relevant to the input domain that are also relevant to the processing required by the specification, i. e., we try to extract conditions from the specification that we, as programmers, would consider relevant to deciding when it is appropriate to perform certain actions. For example, if we think in terms of scanning the input text character by character, then it's possible and reasonable to describe the input in terms of the character currently being scanned and the one immediately preceding it. This approach gives rise to the condition table shown in Table 6. The notation there shows that the values of PrevChar (condition C1) and CurChar (condition C2) constitute conditions relevant to the program's correct operation. In particular, the possible values for C1 and C2 are partitioned into three subsets, the value BL (for BLANK), the value NL, and all other character values, Ot. Any condition may, of course, be undefined (represented in the table by a question mark), and so there are four possibilities to be considered for each condition, yielding 16 predicates in all.

The next step is to check if all these predicates can be satisfied. Clearly the current or previous character may have an undefined value. C2 is undefined when the end of the text is found. C1 is undefined when the current character is the first character in the text. Predicate 16 therefore is satisfied by an empty text. To make this interpretation of undefined values clearer, in later condition tables we will redefine the value set for CurChar to include the pseudo-character ET, designating end-of-text, and not permit C2 to be undefined.

Looking further, at predicates 1 and 2, we immediately see an ambiguity in the specification. It's not clear whether words are separated by a single BLANK or NL character, or whether several such break characters are permitted. If only one is permitted, then predicates 1, 2, 5, and 6 are not satisfiable. Also, if breaks are not permitted after the last word in the text or before the first word, then predicates 4, 8, 13, and 14 can be eliminated as unsatisfiable. Permitting all these predicates to be satisfied seems the most reasonable approach since it provides more flexibility for dealing with various types of input media(e.g., cards or paper tape) and this interpretation subsumes the more restricted case of single break characters, so we adopt it. Of course, the ambiguity should be checked with the specification writer, since our reasoning for permitting data satisfying these predicates may not be valid in the context of the actual use of the program. It can be seen here how preparation of a condition table is a natural way to check a specification for completeness and lack of ambiguity, as well as for identifying characteristics of test data that should be presented to the program during the test phase.

Next we need to check the test predicates for reliability, i. e., if we select data to cover each predicate, will we be forced to select data capable of revealing all errors in an implementation? To answer this question, we must decide whether any conditions relevant to the correct operation of the program are missing from the table and whether value sets have been partitioned appropriately. We must also decide whether the test predicates are independent, i. e., whether the sequence in which test predicates are exercised when processing test data is potentially significant to the correctness of the program. For example, we can see that data chosen to exercise all the predicates in Table 6 will have to include data where words are separated by at least two break characters, but all predicates can be exercised without necessarily having exactly one break character between any words. * Is the occurrence of a break of length one significant to the correct operation of a program? If a program correctly processes text containing breaks of length two or greater, will it necessarily correctly process text containing breaks of length one? What about breaks of length one and two preceding the first word in

*For example, if predicate 9 is exercised followed by predicate 3, a break of length one has been processed, but if the sequence is 9, 1, 3, then the break is two characters long. So length of breaks is represented as a particular sequence of predicates being executed.

the text or following the last word? From experience, we know that errors involving all these sorts of conditions are quite possible, so we will add conditions to force selection of data that check these error possibilities.

To distinguish breaks before the first word, we add a condition "First Word?", which is false until one Ot character is seen and then true. To ensure test data contain breaks of length 1, the condition "Break length" is added; its values refer to the length of a break preceding ET or Ot. Table 7 results; note the conditions are not all independent. When C1 is undefined, C3 is false, etc. Constraints between conditions are indicated below the condition table.

Are the test predicates sufficiently independent now? Do some sequences of test predicates execution correspond to special predicates that should be included? It can be seen that data consisting of at most one word would suffice to exercise all the test predicates. Equally well, all test data could consist of texts containing more than one word. It's highly unlikely that a program will process words correctly for texts several words long but not for texts one word long, or vice versa, although such a program could of course be specially written. We will be content, for now, with noting that the number of words in an input text might be a significant condition for some programs; later when an implementation is available we can decide whether the number of words in the text is a condition for which an error may exist; if so, the condition table can be modified to insure this aspect of the program is thoroughly tested.

Test predicates can be eliminated (i.e., their exercising made unnecessary) if it can be shown that data satisfying the predicates are treated the same by the actual program and from the viewpoint of the specifications. For example, distinguishing between BL and NL in Table 7 is unnecessary. Cursory examination of the programs in either Figure 1 or Figure 2 shows that every time CW is tested for equality with BL it is also tested for equality with NL; the specifications, moreover, do not require different effects depending on whether a BL or NL is the value of a break character. So we can safely partition the value set for CurChar into BL or NL, ET, and Ot and the value set for PrevChar into BL or NL, and Ot without reducing the reliability of this set of test predicates for these particular programs. This will reduce the number of test predicates in Table 7 from 32 to 16 and the number of test runs required from ten (one for each ET condition) to six. This illustrates how the amount of testing can be safely reduced after a set of test predicates has been developed independently of program structure. Proofs of simple program properties can reduce the amount of testing required.

The next part of Naur's specification states:

> "...line breaks (in the output) must be made only where the (input) text has BLANK or NL..."

This clause states a constraint on when the action of outputting a NL character is valid. Since we have already distinguished BLANKs and NL characters in Table 7, no new test conditions are suggested. Examining the clause with respect to Table 7, however, shows that a line break can never be made before the first word in the text unless the first word is preceded by one or more break characters. Naur's program itself violates this clause because it always puts out a line break at the beginning of the output. Undoubtedly, the intent of the clause was not to forbid an initializing line break in the output, but rather to say that a word in the input text must be contained completely on a single line in the output; here is another specification error (failure to correctly express the intent of the designer) readily detected by case analysis.

The remainder of the specification states that "each line is filled as far as possible as long as no line will contain more than MAXPOS characters." We will not construct a condition table for this part of the specification. Such a table would contain conditions describing the length of words, current and future lengths of lines, and possibly, the number of words on a line. The table will show how to exercise a program for all relevant combinations of boundary conditions. Arguments discussing what constitutes a proper set of test predicates for this part of the specification are complex and almost require a paper in itself. We only intend to sketch an approach to test predicate construction here.

4.3 Summary

There are two main features of the method of test data selection presented here--the use of test predicates and the use of a condition table.

Test predicates are the motivating force for data selection. A test predicate is found by identifying conditions and combinations of conditions relevant to the correct operation of a program. Conditions arise first and primarily from the specifications for a program. As implementations are considered, further conditions and predicates may be added. Conditions can arise from many sources, and as they come to mind, they are written down in the condition stub of a table. The columns of the table are then generated and checked. The checking process may suggest further conditions to be added. If the number of predicates becomes excessive, it is better to wait until information from an actual implementation is available to reduce the number of predicates so the impact of the reduction on test reliability can be explicitly assessed.

The major problem in our approach as illustrated so far is that conditions are considered as they spring to mind. This may mean wasting effort on conditions unlikely to be connected with errors in the actual implementation to be tested. Nonetheless, such conditions do test something about a program. The quality of the test predicates cannot be decided at the exact moment of their conception. Test predicate analysis must be carried out in its entirety and then checked for overall reliability before deleting individual test predicates.

The emphasis is on a systematic approach, and the condition table fulfills this purpose by recording conditions and their combinations in an orderly and mechanically checkable fashion. It forces attention toward specifications and what the program should do rather than an

actual program structure and what a program seems to do. It avoids the flaws of testing methods that focus solely on the internal structure of a program, but is not necessarily divorced from a program's internal structure since all program predicates must ultimately be represented in the condition-table if the table is to define a reliable set of test predicates.

The type of reasoning used in selecting test predicates is much like that used in creating assertions, and hence the approach focuses attention on the abstract properties of the program and its specifications. A test predicate analysis of a program may be a practical first step toward program proving with the advantage that both testing and proving could be performed sequentially or in parallel.

5. The Theory of Testing

The methodology illustrated in the preceding section constitutes an informal application of the theoretical concepts defined in Section 1, i. e., Section 4 demonstrates the use of COMPLETE(T,C) RELIABLE(C), and VALID(C), as a means of devising thorough tests when the test data selection criterion, C, consists of a set of test predicates. In this Section, we will point out how our use of this sort of test data selection criterion satisfies these previously defined theoretical concepts.

Since a test predicate is a condition combination in a condition table, selecting data to satisfy a test predicate means selecting data to exercise the condition combination in the course of executing F. Notationally, if a datum $d \in D$ satisfies test predicate $c \in C$ in this sense, then c(d) is said to be true. With this definition in mind, COMPLETE(T, C) is defined as follows:

$$\text{if } T \subseteq D,\ \text{COMPLETE}(T, C) = (\forall c \in C)(\exists t \in T)c(t) \wedge (\forall t \in T)(\exists c \in C)c(t)$$

This definition states that every test predicate belonging to C must be satisfied by at least one $t \in T$, and every t must satisfy at least one test predicate. Both requirements are necessary, since proof of C's validity will require proving the correctness of a program for data that satisfy no test predicate, and hence, are included in no set of test data T.

The examples in Section 2 showed that it is readily possible to choose data that exercise test predicates in an overlapping manner (e. g., see Table 5). This suggests a natural partitioning of the input domain into related equivalence classes, E(C') defined as follows:

Let $C' \subseteq C$.

$$E(C') = \left\{ d \in D \mid (\forall c \in C')c(d) \wedge (\forall c \in C - C')\neg c(d) \right\}$$

i. e., E(C') contains all $d \in D$ that satisfy all and only those test predicates belonging to C'. Note that data belonging to the equivalence class $E(\phi)$ satisfy no test predicate. The relation between these equivalence classes for a C containing three test predicates is shown schematically in Figure 4. Note that data belonging to, for example, $E(C_{12})$ satisfy just test predicates c_1 and c_2, and that if test T contains data belonging to $E(C_{12})$, T need not also contain data belonging to $E(C_1)$ and $E(C_2)$ to satisfy COMPLETE. Note that a complete test, therefore, is equivalent to selecting one datum from a subset of the equivalence classes without selecting any data from $E(\phi)$.

C	$= (c_1, c_2, c_3)$
C_{123}	$= C$
C_{12}	$= (c_1, c_2)$
C_{13}	$= (c_1, c_3)$
C_{23}	$= (c_2, c_3)$
C_1	$= (c_1)$
C_2	$= (c_2)$
C_3	$= (c_3)$

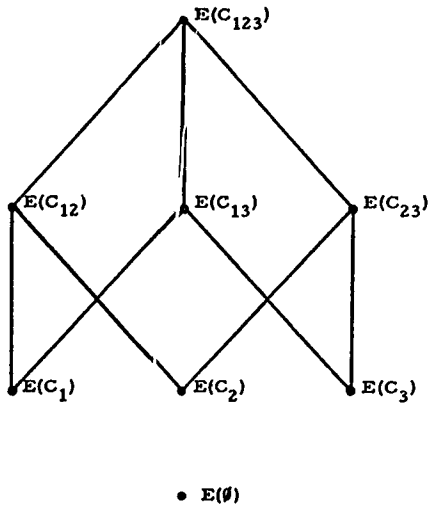

Figure 4

Structure showing relationships between equivalence classes induced by a set of test predicates, C. If C is reliable, the correct processing of data drawn from one equivalence class (e.g., $E(C_{12})$) proves the correctness of the program for data drawn from certain other classes (e.g., $E(C_1)$ and $E(C_2)$), i.e., success propagates downwards in the lattice of equivalence classes. Similarly, failure propagates upward. For example, the incorrect processing of data drawn from $E(C_3)$ implies data drawn from $E(C_{23})$, $E(C_{13})$, and $E(C_{123})$ will also be processed incorrectly.

Of course, not just any set of test predicates will constitute a reliable and valid test data selection criterion. The following relationship can be used, however, to show that C will satisfy the general definition of VALID given in Figure 1.

$$(\forall d \in E(\phi))\ OK(d) \supset VALID(C)$$

This means C is valid if a program is correct for all data satisfying no test predicate. With respect to the condition table technique, this validity criterion requires that constraints among conditions be properly described; every condition combination excluded as impossible must actually be impossible. Moreover, the domain of values associated with some variable must be correctly described, e.g., CW in Section 4 must at most take on the values BL, NL, ET, and Other. Although a non-empty $E(\phi)$ can be viewed as warning that a program can execute a reliable test successfully and still contain errors because it is invalid, it can also be viewed merely as indicating that the correctness of the program for such data is to be assessed by means other than the tests implied by C. Separate verification, for example, by proof, may be easier than verification by testing, or it may be that the data in $E(\phi)$ have already been verified by prior tests. It may be that a conscious decision has been made not to exercise certain test predicates because it is "obvious" the program will correctly process data satisfying them. This can reduce the total testing effort. In any event, the program's correctness for data belonging to $E(\phi)$ is not determined by the tests defined by C.

It may sometimes be impractical to conduct complete tests. In this case, if C is known to be valid because $E(\phi)$ is empty, then test data should be chosen just from equivalence classes that satisfy test predicates likely to be encountered in practice. While such a test will not necessarily be valid, it will still be reliable if C is reliable, and estimates of overall program reliability can be devised by estimating the frequency with which data satisfying unexercised test predicates will be encountered in actual use of the program. This sort of analysis can set a lower bound on the estimated frequency of failure in the program's actual operation, even when a program is tested only incompletely.

The formal definition of RELIABLE(C) implied by the use of test predicates is rather complex, but the essential idea is to define test predicates so success or failure when executing a program F with a particular datum d depends only on what individual test predicates d satisfies, not the particular combination satisfied. This is the sense in which test predicates must be independent. For example, using the definition of C given in Figure 4, if $d_{12} \in E(C_{12})$ and $OK(d_{12})$, then the definition of COMPLETE says that there is no need to test F with data belonging to $E(C_1)$ or $E(C_2)$. Hence $OK(d_{12})$ implies $OK(d_1)$ and $OK(d_2)$, where $d_1 \in E(C_1)$ and $d_2 \in E(C_2)$ if C is reliable. Similarly, $OK(d_1)$ and $OK(d_2)$ implies $OK(d_{12})$, since one complete test could include d_{12} and another, both d_1 and d_2; regardless of which complete test is actually performed, the success of either complete test must imply the success of all others if C is reliable. Note also that the definition of COMPLETE implies that to be reliable, neither the sequence of test predicate execution nor the frequency of execution can be relevant to the successful processing of test data because data are said to satisfy a test predicate whether the data cause the predicate to be exercised one time or many times, and no matter what sequences of predicates are exercised before or after a particular one. If frequency or sequence of test predicate execution is potentially relevant to an error's existence, separate test predicates must be defined to cover all relevant sequencing or frequency possibilities. (This was the reason we added "break length" and "first word?" to the condition table created in Section 4.)

The formal definition of RELIABLE(C), when C is a set of test predicates, therefore is:

$$\begin{aligned} RELIABLE(C) = {} & (\forall C_1 \subseteq C)(\forall C_2 \subseteq C)(\\ & (\exists t_1 \in E(C_1))\ (\exists t_2 \in E(C_2)) \\ & (OK(t_1) \wedge OK(t_2)) \supset \\ & (\forall C' \subseteq C_1 \cup C_2)(C' \neq \phi \supset \\ & (\forall d' \in E(C')\, OK(d'))\ ) \end{aligned}$$

This means that if a program is correct for some datum, d, in equivalence classes other than $E(\phi)$, then it is correct for any data satisfying a (non-empty) subset of the test predicates exercised by d. Note that tests containing only data satisfying a large number of test predicates are more powerful in the sense that data selected from only a few equivalence classes suffices to demonstrate program correctness. Such tests are useful when the probability of finding an error is deemed low, as in regression testing [Elmendorf (1969)]. Tests using data satisfying only a few test predicates can fail for fewer reasons and so are useful in either localizing a failure or in attempting to identify as many different errors as possible in a set of test runs. Minimizing the cost of testing requires judgement in deciding which equivalence classes to choose from in satisfying COMPLETE(T, C).

We discussed earlier why proving validity need not be difficult. Proving reliability is harder. Such a proof suffers from the same problems as direct proofs of program correctness, insofar as insuring the correctness of the proof and axiomatization of the run-time environment is concerned. A proof of reliability may be somewhat easier than a direct proof of program correctness, however, since the proof can assume the program works correctly for some data. The principal problem in the proof is to show that data satisfying each test predicate are treated in sufficiently the same way that successfully exercising each test predicate once is sufficient to imply successful execution for all data satisfying the predicate. At a minimum, this requires understanding all conditions relevant to the successful processing of a test datum, but it does not require proving that the test datum will be successfully executed--that is proved by the success of a test run. This shows that proving test reliability is the induction step in an inductive proof of program correctness using the fundamental theorem; the test itself is, of course, the basis step.

Defects in a proof of reliability, aside from simple logical errors, are most likely to stem

from failure to find all conditions relevant to the successful processing of data. For example, the set of test predicates described by a condition table will fail to be reliable if the domain of values associated with some variable is not correctly partitioned into relevant classes of values (e.g., the test predicates developed in Section 4 would be unreliable if the only relevant values of CW were not BL, NL, ET, and Other, i.e., if these were not the only values relevant when certain actions are to be performed) or if some variable or condition were completely missing from the table (e.g., the predicate fill + bufpos = MAXPOS). Note the difference here between a reliability error and a validity error. Reliability requires partitioning the value set for CW correctly into sets of values "treated the same" by the program. Validity requires the postulated value set for CW include all possible values for CW.

It's impossible to formulate general necessary conditions for reliability since reliability means that all errors in a program will be detected by a complete test. If a program in fact has no errors, any set of test predicates is reliable. For example, exercising all statements is not a necessary condition for reliability as long as unexercised statements are never responsible for an error. But suppose that test predicates are defined so data belonging to a particular equivalence class sometimes exercise a particular statement and sometimes do not. Then proving the reliability of these test predicates will be harder because it must be shown that the statements not exercised by some data in the equivalence class never cause an error. Otherwise, if an error is associated with the occasionally executed statement, some data in the class will execute successfully and some will not, and this contradicts RELIABLE(C). So, in general, we can only describe certain characteristics that if not satisfied either make a proof of reliability harder or make it more likely the test predicates are not reliable for the program to be tested. For example, it appears that a set of test predicates must at least satisfy the following conditions to have a reasonable chance of being reliable:

1) every individual branching condition in the program must be represented by an equivalent* condition in the table;
2) every potential termination condition in the program [Sites(1974)] e.g., overflow, must be represented by a condition in the table;
3) every variable mentioned in a condition must have been partitioned correctly into classes that are "treated the same" by the program;
4) every condition relevant to the correct operation of the program that is implied by the specification, knowledge of the program's data structures, or knowledge of the general method being implemented by the program must be represented as a condition in the table;
5) the test predicates must be independent, e.g., all data satisfying a particular test predicate must exercise the same path in the program and test the same branch predicates.

*This means verifying that the conditions represented in the condition table are being accurately tested inside the program; this check is needed to insure that a table predicate, e.g., "X, Y, and Z are all equal" is not actually evaluated in the program as (X + Y + Z)/3 = X.

Note that constraints 1, 2, and 3 are satisfied only with knowledge of the details of a program's implementation. Satisfying constraint 5 requires knowing something of the program's internal structure. Verifying constraint 4 may require both internal and external knowledge of a program. Undoubtedly, more constraints than these five need to be satisfied to insure reliability, but this is a subject for future work.

Clearly, satisfying all these constraints can lead to lots of test predicates requiring a large set of test data for a complete test. If an unreasonable amount of test data is required, a judicious combination of direct program proving and empirical judgement can reduce size of a complete test. As long as test predicates are eliminated only after all conditions potentially relevant to the correct operation of the program have been identified, any reductions in test validity are at least being made consciously rather than unwittingly.

6. Summary

In this paper, we have examined some fundamental questions and problems concerning program testing:

1. What is the proper role of testing in program development? How does testing compare to program proving as a means of improving software reliability?
2. What are the weaknesses of testing and to what extent can they be overcome?
3. What is a systematic method for selecting test data and how can the method be evaluated as to its reliability and validity?
4. Is there a theory of testing that can point the way to improved methods and principles of testing?

In answering these questions we have shown the following.

1. The Role of Testing - Testing has an essential role in program development if for no other reason than the pervasiveness of case analysis in analyzing requirements, defining specifications, implementing programs, and proving program correctness. We have discussed how the practice of attempting formal or informal proofs of program correctness is useful for improving reliability, but suffers from the same types of errors as programming and testing, namely, failure to find and validate all special cases (i.e., combinations of conditions) relevant to a design, its specifications, the program, and its proof. Neither testing nor program proving can in practice provide complete assurance of program correctness, but the techniques complement each other and together provide greater assurance of correctness than either alone.

2. Test Weaknesses - The weakness of testing lies in concluding that from the successful execution of selected test data, a program is correct for all data. Criteria for test data selection based solely on internal program structure are clearly too weak to insure confidence in the results of a successful test, since such testing methods are

too weak to necessarily reveal all design errors or many types of construction errors. Reliability is the key to meaningful testing and lack of reliability is the reason for the current weakness of testing as a method for insuring software correctness. Further work is needed to show when a test is reliable.

3. Test Methodology - We believe most software errors result from failing to see or deal correctly with all conditions and combinations of conditions relevant to the desired operation of a program. An effective methodology for reliable program development must focus on uncovering these conditions and their combinations. This is the motivation for our use of condition tables--it makes it easier to find and analyze condition combinations. Experimental use of condition tables and further study of the method is needed, however, before its full benefits can be realized in practice. In any event, a systematic approach to test data selection is clearly necessary to obtain effective test data with reasonable effort. Most systematic methods proposed to date have been based solely on knowledge of a program's internal structure, and yet such knowledge is insufficient by itself to yield adequately reliable tests.

4. Theory of Testing - We have made a start toward a theory of testing by defining some basic properties of thorough tests--reliability, validity, and completeness. We have also developed a specific version of the theory in which the test data selection criterion consists of sets of test predicates organized in a condition table. Other theoretical developments could proceed by defining other types of test data selection criteria.

The concept of proving test reliability, validity, and completeness and then testing a program (rather than attempting to prove the program correct directly or using a non-rigorous approach to testing) is important for software reliability. Presumably, every piece of software undergoes some type of testing, yet the fundamental techniques of test data selection have not previously been as carefully and critically analyzed as have some less widely used techniques such as proof of correctness. We know less about the theory of testing, which we do often, than about the theory of program proving, which we do seldom. This paper is a step toward redressing this imbalance.

References

Boehm, B. Software and its impact: a quantitative assessment. Datamation 19(May 1973), 48-59.

Boehm, B. W. et. al. Some experience with automated aids to the design of large-scale reliable software. In this proceedings, April 1975.

Buxton, J. N. and Randell, B. Software Engineering Techniques, Scientific Affairs Division, NATO, Brussels 39, Belgium, April 1970.

Dahl, O. -J., Dijkstra, E. W., and Hoare, C. A. R. Structured Programming. Academic Press, New York, 1972.

Elmendorf, W. R. Controlling the functional testing of an operating system. IEEE Trans. Sys. Sci. and Cyb., SSC-5, 4 (October 1969), 284-289.

Hetzel, W. C. Program Test Methods. Prentice-Hall, Englewood Cliffs, N. J., 1973.

Hansen, P. B. Testing multiprogramming systems. Software Practice and Experience 3, 2 (April-June 1973), 145-150.

King, P. J. H. The interpretation of limited entry decision table format and relationships among conditions. Computer Journal 12, 4 (November 1969), 320-326.

London, R. L. Software reliability through proving programs correct. IEEE Int. Symp. on Fault Tolerant Computing, March 1971.

Naur, P. Programming by action clusters. BIT 9, 3 (1969a), 250-258.

Naur, P., and Randell, B. (eds.) Software Engineering. Scientific Affairs Division, NATO, Brussels 39, Belgium, January 1969b.

Pooch, U. W. Translation of decision tables. Computing Surveys 6, 2 (June 1974), 125-151.

Poole, P. C. Debugging and testing. In Bauer, F. L. Advanced Course on Software Engineering, Springer-Verlag, New York, 1973, 278-318.

Rubey, R. J. et. al. Comparative Evaluation of PL/I., AD669096, April 1968.

Sites, R. L. Proving that computer programs terminate cleanly. Stanford Univ., Computer Sci. Dept., STAN-CS-74-418, May 1974.

Stucki, L. G. and Svegel, N. P. Software Automated Verification System Study, McDonnell Douglas Astronautics Co., AD-784086, January 1974.