August 26, 1970 IBM Technical Report TR 00.2089

Automated Design of Program Test Libraries

by

William R. Elmendorf

Editor's Note:

Just three years after writing the paper on equivalence class testing, Bill recommends that we need to add more rigor to the testing process. In this paper seven more breakthroughs are described.

1. The first step in the process is to create a Cause-Effect Graph of the logic. This is the first application of Model Based Testing in software.

2. He recognizes that the act of graphing the specification actually is a test of the specification itself – i.e., we must test the specification, not just the code derived from the specification.

3. He describes how to create a set of tests that are necessary and sufficient to test the code from a black box perspective. The process eliminates redundancy in the test library while maximizing coverage.

4. He extends the concepts from the hardware version of the algorithms to cover software specific issues. The issue is constraints on the inputs – e.g., exclusive, one and only one. Today these are called preconditions and post conditions in defining use cases. The same concepts are also applied today in design by contract.

5. He introduces the concept of "Untestable" variations. These are variations which, due to the constraints and overall logic, cannot be sensitized to an observable point. The result is that you need to insert diagnostic probe points into the code in order to make sure that you got the right answer for the right reason. The process guides you to where you need to insert the diagnostics.

6. The test case design rules do not just address what input combinations to test. They also identify the expected results. This is the first test case design "Oracle".

7. He automates the test design process in a tool called TELDAP – TEst Library Design Automation Program. This is one of the first software test case design tools. It addresses the problem of ensuring that the set of tests are not only optimized but thoroughly documented.

When the process and the tool were applied to projects they resulted in a 40% to 50% reduction in the cost of the functional test effort while increasing coverage by 30% to 40%. This was as compared to equivalence class testing.

Richard Bender
rbender@BenderRBT.com
October 2006

ABSTRACT

Given the task of developing a test library to exercise a program, the tester organizes his job into three phases: (1) identifying the functions of the program, (2) designing a test library that covers these functions, and (3) writing and debugging the test cases which make up this library. This paper describes an experimental technique for automating and disciplining the second phase.

A two-stage process is presented. First, the functions of the subject program are structured in the form of a Boolean graph. This graph yields an unambiguous definition of the functional variations eligible for testing. Second, the graph is used as input to an algorithm which synthesizes those test patterns that will exercise all functional variations and will distinguish a good program from a bad one. This algorithm is implemented in an APL\360 program named TELDAP (TEst Library Design Automation Program).

Two libraries are defined by this tool. One seeks minimum functional coverage in each test and is appropriate for new-function testing. The other seeks the minimum number of tests and is best suited to regression testing of old function. The documentation for each library identifies the inputs to be invoked or suppressed in each test, the outputs expected from each test if correctly executed, and the faulty functional variations detectable by each test.

Boolean graph
Test pattern generation
21 Programming
22 Reliability-Testing

INTRODUCTION

Typically, the development of program test libraries proceeds something like this. A set of program specifications, together with a program listing, serve as input. They are analyzed for the purpose of generating an exhaustive list of the product's functions. A set of tests is then defined to exercise the identified functions. Finally, when these tests have been written and debugged, the test library is ready for use.

Throughout this process manual effort and artistic judgment predominate, and although the resource investment is substantial, the resulting test library is generally inefficiently structured and inaccurately documented. This library, or a subset of it, is executed many times during the life of the product; therefore, its inefficiency repeatedly absorbs unnecessary human and machine resources. Furthermore, since this library is the fundamental tool for assessing program quality, its documentation errors color this assessment.

What can be done to improve this picture? At the leading edge of the process, work is in progress on the development of formal specification languages and on experimentation with them as a substitute for natural languages. SPEC[3], APL\360[5] and ULD[11] have been considered in this role. Methods of formal path analysis within programs are also being explored. The works of Bender and Pottorff[2], Herman and Pearson[10] and Hess[8, 9] are illustrative. At the trailing edge of the process, test cases are being mechanically generated to take advantage of standardized routines.[12] Where formal syntactical (structural) rules exist for a programming language and we are content to ignore the semantics (the meaning) of the language structures, then the whole process can be automated[6]. However, the resultant test library will generally be neither necessary nor sufficient. It will not be necessary because of the redundant testing of certain functions; it will not be sufficient because there is no assurance that all interactive mixtures of function will be tested.

The study this paper is reporting has been aimed at the middle of the process -- the disciplined design of a test library given a rigorous description of the functions to be tested. The objective is more precisely stated as follows:

> Given an unambiguous and non-redundant description of a set of program functions, design the test library which is necessary and sufficient to exhaustively exercise these functions and distinguish a good program from a bad one.

The study has yielded an experimental process that is described in succeeding sections. The section on Input considers the use of a Boolean graph as the vehicle for rigorously describing the structure of the input functions. The section on Process addresses the algorithm that governs the way in which the graph is systematically massaged to

synthesize the design of each test library. (Two distinct libraries are synthesized.) The section on Output identifies the kinds of output information which constitute a complete test library design.

INPUT

The fundamental step in the software testing process is the identification of those functional elements that are eligible for testing. This identification occurs in two stages: (1) the interpretation of the specifications and the program to deduce the functions described in the former and implemented in the latter, and (2) the dissection of these functions into the functional elements whose composite totally, but without redundancy, represents the functional capability of the product. The identified elements serve three purposes: first, to give perspective to the test objectives; second, to establish the context of the test plan; and third, to act as a yardstick for measuring test coverage.

Cause and Effect Graph

Given a set of functions which represents our best interpretation of the true meaning of the specifications and of the program, we express the structure of these functions in the form of a Boolean graph. Since a function can be defined as a logical cause-and-effect relationship, this graph is called a cause-and-effect graph. It is generated as follows:

- List all the directly invokable causes down one side of a sheet of paper and all the directly observable effects down the other side.

- Join cause to effect by means of a structure of logical connectives that represents the functional relationships decreed by the analyst's interpretation of the specifications and program.

- The eligible connectives are AND, NAND, OR, NOR, NOT and DIRECT. Their notations, as used in this paper, are A, A̲, O, O̲, N and D, respectively. The DIRECT connective expresses the simplest relationship: a single cause joined to a single effect with no negation or interaction with other causes.

- Wherever a known cause leads to an unknown effect or a known effect is not traceable to a known cause, a dummy node is inserted in the graph to represent the unknown. Ultimately this uncertainty must be resolved in order to write the test case that invokes the cause or observes the effect. In the interim, the explicit inclusion of the unknown node in the graph permits the test library design to proceed.

- At this time only combinations of causes, not permutations, can be portrayed. Although the latter is not quantitatively significant in most software testing, it is a worthy candidate for follow-on study.

There are two additional considerations in connection with this graph: range values and cause constraints. When specifications describe both the functional meaning of a parameter as well as the range of acceptable values for this parameter, we have a problem representing both kinds of information in a common graph. On the other hand, if we separate them, then two libraries would have to be designed, one to test functional capability and the other to test the range of this capability. To avoid this we treat certain range values as pseudo-functions that are subordinate to the functions to which they apply.

The invokable causes may be constrained in one of these ways:

- Causes may be mutually-exclusive, meaning that at most one of the set may be invoked in anyone test.

- Causes may be all-inclusive, meaning that at least one of the set must be invoked in every test.

- Causes may be both mutually-exclusive and all-inclusive, meaning that one and only one of the set must be invoked in every test.

A particular cause can be party to only one such constraint. These constraints are indicated in the cause-and-effect graph by the letters E, I and U, respectively.

Figure 1 presents the cause-and-effect graph for the illustrative set of functions at the top of this figure. In the interest of simplicity only a subset of the eligible connectives and constraints is incorporated into this example. A more comprehensive example appears in Appendix A.

IF cause 1 OR cause 2 THEN effect 4.
IF NOT BOTH cause 2 AND cause 3 THEN effect 5.
Causes 1 and 2 are mutually-exclusive.

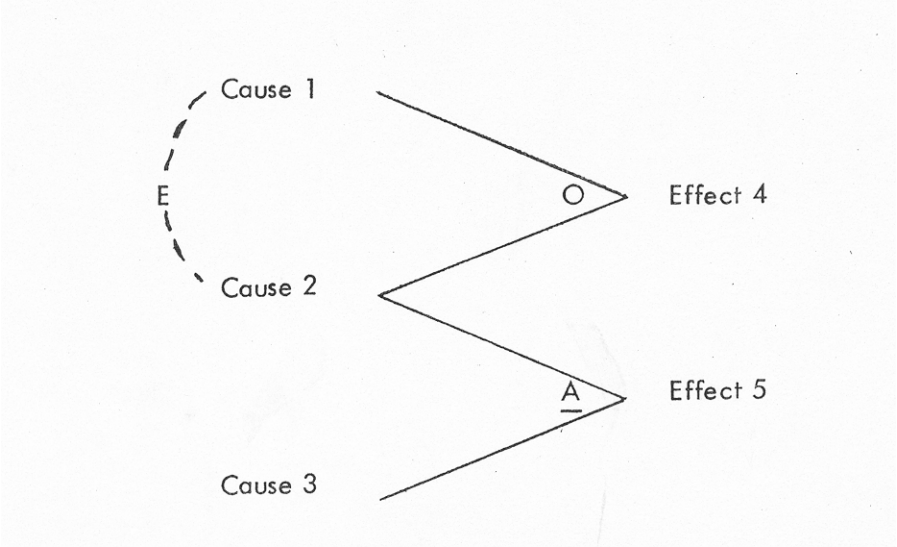


Figure 1. A sample set of program functions and the corresponding cause-and-effect graph.

Note that the cause-and-effect graph of Figure 1 includes explicit expression of the lattice form of structure in which a single cause leads to multiple effects. Since the graph is the input to the library design algorithm, this lattice structure is of necessity considered in the design process. In contrast to this is the tree structure of traditional functional variation lists which does not explicitly state the commonality of a single cause leading to multiple effects. Instead, it is expressed implicitly by replicating the common cause in all of the affected variations. When such a functional variation list is the input to the library design process, it is the exception, rather than the rule, for this implied commonality to be recognized.

Functional Elements

The functional elements of a cause-and-effect graph are the nodes together with the line segments connecting them. Each node represents a cause, an effect, or, in turn, both. Each segment expresses a primitive dependency relationship -- that is, a conditional statement (IF...THEN...) involving a single cause and a single effect.

Segments which converge on a common node represent an interactive dependency -- that is, a conditional statement involving multiple causes and a single effect. On the other hand, segments which diverge from a common node represent the mutual dependency of multiple effects on a common cause.

Each node can assume one of two states: 0 or 1. “0” means that the cause/effect is absent; while “1” means that it is present. If the state of a node is irrelevant or unknown, this is

noted in test patterns by an “X”. The nodes are arbitrarily numbered; the only restrictions are that the numbers must begin at 1 and be contiguous.

What are the functional elements that can fail and are therefore eligible for testing? In a previous paper[4] the "distinctive functional variation” was proposed as this element. A function was defined earlier as a logical cause-and-effect relationship. Each variant of a function is such a relationship with the state (presence or absence) of each cause and effect specified. Consequently, every line in a truth table for a particular function is a variant of that function. Distinctive variants are those at the finest possible level of resolution -- that is, those which cannot be reduced to more elementary variations because they contain some unique cause or effect state not represented in the more elementary ones.

Since these distinctive functional variations encompass both states of all causes and all effects, they comprise the necessary and sufficient scope of testing to detect all functional errors. Put in the context of the cause-and-effect graph, each primitive dependency (each segment) along with each interactive dependency (each convergence of segments) is a distinctive functional variation eligible to be tested.

Faulty variations are traceable to faulty segments: one variation to one segment for primitive variations and one variation to multiple segments (anyone of which may be faulty) for interactive variations. Since all variation failures can be mapped into segment failures, either of these could be addressed by the test library. For reasons of efficiency and convention we use the variations.

Statements

The cause-and-effect graph is entered into the test library design process via a series of input statements drawn from the types shown in Figure 2. The causes and effects are filled in using the appropriate node numbers from the graph. The total node count is entered through the NODES statement; this statement must precede all others. Figure 3 shows that set of statements which embodies the cause-and-effect graph of Figure 1.

NODES number
causes AND effect
causes OR effect
causes NAND effect
causes NOR effect
cause DIRECT effect
cause NOT effect
EXCLUSIVE causes
INCLUSIVE causes
EXCLUSIVE Δ INCLUSIVE causes
INVOKABLE causes
OBSERVABLE effects

Notes:
- Upper-case words are required keywords.
- Lower-case words are replaced by numeric scalars or vectors.

Figure 2. The types of TELDAP input statements.

```
NODES 5
1  2  OR  4
2  3  NAND  5
EXCLUSIVE 1  2
INVOKABLE  1  2  3
OBSERVABLE  4  5
```

Figure 3. The TELDAP input for the sample problem.

PROCESS

Simply stated, the test libraries are designed by systematically tracing through the cause-and-effect graph and synthesizing those patterns of invokable causes which ensure that every faulty variation will lead to an erroneous pattern of observable effects. The library design algorithm described later in this paper assumes only single occurrences of faulty variations. It is probable that most multiple fault situations will, in fact, be detected, but it is theoretically possible for some of them to cancel each other out and go undetected.

A “test pattern” identifies those causes to be invoked, those to be suppressed and those to be ignored. In addition, it identifies the observable effects that can be expected if no faulty variations are encountered. Test patterns can be categorized as "feasible”, "meaningful" or "unique”.

* A test pattern is feasible as long as it does not violate the exclusive and inclusive constraints.

* A test pattern is meaningful if it serves to distinguish a good program from a bad one.

* A test pattern is unique if it detects a distinct set of faulty variations.

Test libraries can be similarly categorized as feasible, meaningful or unique to reflect the type of test patterns they contain. Two useful subsets of the unique library can be defined as:

* The set of unique tests which covers all detectable faults with minimal redundancy and with minimal coverage per test is the new-function test library.

* The set of unique tests which covers all detectable faults with the near –minimal number of tests is the old -function test library.

Brute-Force Approach

Let us now consider a systematic procedure for determining sets of feasible, meaningful, unique, new-function and old-function test patterns. Such a procedure can be illustrated by designing the test libraries for the cause -and -effect graph of Figure 1.

* First, identify the distinctive functional variations and number them for subsequent reference. This list appears in Figure 4.

* Second, list all possible combinations of the invokable causes. In Figure 1 there are three of these causes, each of which can be invoked, suppressed or ignored. Consequently, there are 3^3 or 27 possible test patterns to be considered.

* Third, strike out those patterns that are infeasible due to exclusion or inclusion constraints on the invokable causes. Figure 5 lists the nine test patterns which violate the exclusion constraint between causes 1 and 2. The remaining test patterns (27-9 = 18) comprise the feasible test library.

* Fourth, within the context of each feasible test pattern determine which variation faults will propagate through the cause-and-effect graph to generate erroneous effects. Enter these faults in a "functional matrixl"[4] of the type shown on the right side of Figure 6. A “T” indicates that a fault in this variation is detected by this test.

* Fifth, eliminate those patterns in which all of the effects are indeterminate since this makes it impossible to judge which effects are erroneous. Also eliminate those patterns which, though determinate, do not detect any variation faults.

Figure 5 lists the indeterminate patterns and Figure 6 the ineffectual patterns for our sample problem. Since there are eight such patterns, we are left with 18-8 or 10 tests in the meaningful test library.

* Sixth, arbitrarily eliminate all but one of each set of equivalent patterns – equivalent because they detect the same faulty variations as evidenced by the functional matrix. The discarded equivalent patterns are also listed in Figure 6. The remaining test patterns (10-2 = 8) comprise the unique test library.

* Seventh, by trial and error select the set of unique tests that minimizes redundancy and also disperses the fault coverage across as many tests as possible without sacrificing any of that coverage. This is the new-function test library documented in Figure 7.

* Eighth, again by trial and error choose the set of unique tests which compresses the fault coverage into as few tests as possible, also without sacrificing coverage. This is the old -function test library documented in Figure 7.

1. If cause 1 present then effect 4 present.
2. If cause 2 present then effect 4 present.
3. If causes 1 and 2 absent then effect 4 absent.
4. If cause 2 absent then effect 5 present.
5. If cause 3 absent then effect 5 present.
6. I f causes 2 and 3 present then effect 5 absent.

Figure 4. List of distinctive functional variations for the sample problem.

Test Patterns

	Cause 1	Cause 2	Cause 3	Effect 4	Effect 5
Infeasible	X	1	X	1	X
	X	1	0	1	1
	X	1	1	1	0
	1	X	X	1	X
	1	1	X	1	X
	1	X	0	1	1
	1	X	1	1	X
	1	1	0	1	1
	1	1	1	1	0
Indeterminate	X	X	X	X	X
	X	X	1	X	X
	0	X	X	X	X
	0	X	1	X	X

Figure 5. The infeasible and indeterminate test patterns for the sample problem.

	Test Patterns					Functional Matrix					
	Cause 1	Cause 2	Cause 3	Effect 4	Effect 5	Var 1	Var 2	Var 3	Var 4	Var 5	Var 6
Ineffectual	X	X	0	X	1						
	X	0	X	X	1						
	X	0	0	X	1						
	0	X	0	X	1						
Equivalent	0	0	0	0	1			T			
	0	0	X	0	1			T			
	1	0	0	1	1	T					
	1	0	X	1	1	T					

Figure 6. The ineffectual and equivalent test patterns for the sample problem.

	Test Patterns					Functional Matrix					
	Cause 1	Cause 2	Cause 3	Effect 4	Effect 5	Var 1	Var 2	Var 3	Var 4	Var 5	Var 6
New-Function Test Library	X	0	1	X	1				T		
	0	0	X	0	1			T			
	1	0	X	1	1	T					
	0	1	0	1	1		T			T	
	0	1	1	1	0		T				T
Old-Function Test Library	0	0	X	0	1			T			
	0	1	0	1	1		T			T	
	0	1	1	1	0		T				T
	1	0	1	1	1	T			T		

Figure 7. The test library composition for the sample problem.

This brute -force procedure illustrates the distinctions between the different kinds of test libraries and shows that relatively few tests (4) out of all the combinatorial possibilities (27), and all the feasible tests (18), will exhaustively exercise the functions of this product. Clearly, the brute-force approach is impractical for solving problems of real-world magnitude. A significantly more efficient technique has been implemented in an experimental program called TELDAP (TEst Library Design Automation Program). A description of this program follows.

TELDAP

TELDAP's predecessors were techniques for generating test patterns for the logic circuits in hardware. The application of these techniques, as they stand, to software testing is precluded by the following shortcomings:

1. Feasibility constraints on the cause patterns, such as those imposed by mutually-exclusive and all-inclusive causes, are not recognized.

2. Invokable causes are not distinguished from primitive (primary) causes. This distinction is needed to avoid specifying test inputs at a lower cause level than necessary.

3. Observable effects are not distinguished from end (primary) effects. This distinction is needed to avoid the propagation of faults to higher effect levels than necessary, which, in turn, would require specification of unnecessary input states.

4. In hardware testing the elements of the Boolean graph that may be "stuck-at-l" or "stuck-at-0" are the potentially faulty elements. However, in software testing the Boolean graph is an intermediary for expressing the logical structure of the functions to be tested, and the functional significance of faulty graph elements is not readily apparent. Therefore, it is preferable that the distinctive functional variations be considered the potentially faulty elements and that determining the truth or falsity of these variations be the purpose of the test library.

5. A test library that emphasizes minimal coverage per test while avoiding unnecessary redundancy is not defined. This emphasis is advisable in a library to be used in the early stages of new-function testing. It is also useful in localizing problems found during old-function testing.

6. It would not be practical to design test libraries for cause-and-effect graphs of more than 100 variations because of memory space requirements. Yet problems exceeding this size are common in software testing.

Nevertheless, since there is considerable overlap of hardware and software needs, a hardware method[7] was adopted as the starting point for experimentation in the software environment. This method was selected because of its simplicity, its clarity of description and its orientation toward programmed implementation. However, it was soon found to be prohibitively expensive, even for experimental purposes, and that algorithm was abandoned in favor of one which uses the "path sensitizing"[1] approach.

A precise APL\360 description of the algorithm implemented in TELDAP is given in Appendix B. Its general organization is:

1. The input statements are translated into notations in an internal cause-and-effect matrix.

2. The distinctive functional variations which represent the entire cause-and - effect graph are identified.

3. The path(s) from the causes to the observable dependent effects are defined.

4. The cause/effect states that sensitize these paths so that every detectable fault is propagated to an observable effect are synthesized. (An undetectable fault is noted

by the absence of a T in the appropriate column of the functional matrix generated in step 7.)

5. Those causes /effects whose states are implied by a particular sensitized path, although not immediately within that path, are determined.

6. The resulting test patterns are then compared with each other to eliminate those that are totally contained within other patterns. The patterns that survive this process cover all detectable faults in the maximal number of test patterns, excepting those that are wholly redundant. This is the new-function test library.

7. The functional matrix for this set of test patterns is generated by determining which patterns synthesized in steps 4 and 5 appear in each surviving pattern of step 6.

8. Now the patterns from step 6 are merged, wherever consistency permits, to minimize the number of discrete patterns without sacrificing fault coverage. These merged patterns cover all the detectable faults in a near -minimal number of test patterns. This is the old –function test library.

9. The functional matrix for this set of test patterns is generated by determining which patterns synthesized in steps 4 and 5 are contained within the patterns formed in step 8.

TELDAP has successfully eliminated the first five drawbacks discussed earlier. Regarding the sixth one, the picture is clouded. The current version of TELDAP would require about 35 minutes of Model 50 CPU time and 20K bytes of storage for a 100 variation graph. The time requirement would be reduced significantly (probably by an order of magnitude) by translating the APL program into a compilable language. (Batch execution is adequate.) Given this improvement, the time and space requirements for a 100 variation graph would be quite modest. However, both requirements increase roughly as the square of the number of variations; therefore, for problems of 500 to 5000 variations the TELDAP algorithm, as it stands, may be economically impractical. However, there is reason for optimism that a solution to the performance problem is within our reach. Improvements in the existent algorithm are possible; complexity was consciously avoided in the prototype version since proof of feasibility was the prime objective. It is also possible that a fundamentally more efficient algorithm will emerge from the current work of hardware logicians. Over and above these potential improvements in the algorithm is the generally improved economic picture due to advances in the speed and memory size of computers for a given dollar investment.

OUTPUT

TELDAP produces four kinds of output information:

1. A list of the distinctive functional variations eligible for testing.

2. A pattern of the causes to be invoked or suppressed for each test.

3. A pattern of the effects to be observed in a correctly executed test.

4. A pattern of the faulty variations potentially responsible for a failing test.

The first type of information is generated once at the outset of each TELDAP execution. The remaining three kinds of information are the output for each of two test libraries designed by TELDAP.

One of these libraries is oriented toward the testing of new error-prone functions and therefore emphasizes minimal fault coverage within each test. The test patterns and the functional matrix for this library are derived from Steps 6 and 7 of the TELDAP process described in the preceding section.

The focal point of the other library is regression testing of old and presumably error-free functions; consequently, it seeks maximum fault coverage within each test. Steps 8 and 9 of the TELDAP process are the source of the test patterns and functional matrix for this library.

Figures 8, 9 and 10 illustrate the format of the output documentation. A given test is recorded in corresponding rows of the three matrices that represent each library. The data in these figures is for the sample problem defined by the cause-and-effect graph of Figure 1.

FUNCTIONAL VARIATIONS:

1. IF CAUSE(S) 1 2 ABSENT THEN EFFECT 4 ABSENT
2. IF CAUSE(S) 1 PRESENT THEN EFFECT 4 PRESENT
3. IF CAUSE(S) 2 PRESENT THEN EFFECT 4 PRESENT
4. IF CAUSE(S) 2 3 PRESENT THEN EFFECT 5 ABSENT
5. IF CAUSE(S) 2 ABSENT THEN EFFECT 5 PRESENT
6. IF CAUSE(S) 3 ABSENT THEN EFFECT 5 PRESENT

Figure 8. The distinctive functional variations defined by TELDAP.

THIS LIBRARY CONTAINS 5 TESTS DOCUMENTED IN CORRESPONDING ROWS OF THE FOLLOWING THREE MATRICES:

INVOKABLE CAUSE PATTERNS:
1 2 3

0 0 X
1 0 X
0 1 1
X 01
0 1 0

EACH ROW IS A TEST; EACH COLUMN IS AN INVOKABLE CAUSE
INTERPRET ELEMENTS AS FOLLOWS:

1: INVOKE THIS CAUSE IN THIS TEST
0: SUPPRESS THIS CAUSE IN THIS TEST
X: INVOKE OR SUPPRESS THIS CAUSE IN THIS TEST

OBSERVABLE EFFECT PATTERNS:
4 5
0 1
1 1
1 0
X 1
1 1

EACH ROW IS A TEST; EACH COLUMN IS AN ORSERVABLE EFFECT
INTERPRET ELEMENTS AS FOLLOWS:

1: THIS EFFECT SHOULD OCCUR IN THIS TEST
0: THIS EFFECT SHOULD NOT OCCUR IN THIS TEST
X: THIS EFFECT MAY OR MAY NOT OCCUR IN THIS TEST

FUNCTIONAL MATRIX:
1 2 3 4 5 6
T
  T
    T T
      T
    T T

EACH ROW IS A TEST; EACH COLUMN IS A FUNCTIONAL VARIATION
INTERPRET ELEMENTS AS FOLLOWS:

T: THIS FAULTY VARIATION DETECTED BY THIS TEST
BLANK: THIS FAULTY VARIATION NOT DETECTED BY THIS TEST

Figure 9. The new-function test library designed by TELDAP.

THIS LIBRARY CONTAINS 4 TESTS DOCUMENTED IN CORRESPONDING ROWS OF THE FOLLOWING THREE MATRICES:

INVOKABLE CAUSE PATTERNS:
1 2 3

0 0 1
1 0 X
0 1 1
0 1 0

EACH ROW IS A TEST; EACH COLUMN IS AN INVOKABLE CAUSE
INTERPRET ELEMENTS AS. FOLLOWS:

1: INVOKE THIS CAUSE IN THIS TEST
0: SUPPRESS THIS CAUSE IN THIS TEST
X: INVOKE OR SUPPRESS THIS CAUSE IN THIS TEST

OBSERVABLE EFFECT PATTERNS:
4 5

0 1
1 1
1 0
1 1

EACH ROW IS A TEST; EACH COLUMN IS AN OBSERVABLE EFFECT
INTERPRET ELEMENTS AS FOLLOWS:

1: THIS EFFECT .SHOULD OCCUR IN THIS TEST
0: THIS EFFECT SHOULD NOT OCCUR IN THIS TEST
X: THIS EFFECT MAY OR MAY NOT OCCUR IN THIS TEST

FUNCTIONAL MATRIX:
1 2 3 4 5 6
T T
T
T T
T T

EACH ROW IS A TEST; EACH COLUMN IS A FUNCTIONAL VARIATION
INTERPRET ELEMENTS AS FOLLOWS:

T: THIS FAULTY VARIATION DETECTED BY THIS TEST
BLANK: THIS FAULTY VARIATION NOT DETECTED BY THIS TEST

Figure 10. The old-function test library designed by TELDAP.

SUMMARY

The focal point of this study is the process by which we develop program test libraries. As mentioned earlier, it makes no contribution to either the first step of the process where specifications and program listings are translated into functions, or to the final step where actual test cases are created. Its primary concern is the technology which promises to improve the design process that occurs between these two steps.

The theme of the improvements can be summarized as the substitution of discipline for art and machines for man. These improvements are twofold:

1. Introduction of the cause-and-effect graph as a vehicle for explicitly portraying the structure of a set of program functions.

2. Adaptation of hardware-oriented, mathematically-based test pattern generation techniques to permit their application to program testing.

The first item means that the identification of the functional variations eligible to be tested will be less artistic and partially mechanical. It is not yet evident which effort is greater -- creation of the cause-and-effect graph or today's manual generation of functional variation lists. Certainly the Boolean graph route is more systematic and rigorous and produces a set of variations that is a more reliable yardstick for measuring test coverage. Neither route eliminates the need for experience and judgment in the interpretation of frequently ambiguous, redundant and imprecise natural language specifications and in the analysis of often complex program logic with its obscure relationship to the functions implemented in it.

The second improvement promises the automation of test library design yielding these benefits:

a. More compact test libraries.

b. More accurate test library documentation, particularly in the functional matrix. (Until now functional matrices seldom recorded the “free” coverage achieved by tests outside their assigned domain. Now all of this free coverage is documented.)

c. Potentially a faster and cheaper process.

At present TELDAP is bracketed by manual steps which compromise its potential benefits. This makes the future of such a tool highly dependent on the future of formal specification languages and test case generators. Given the former, the input to TELDAP could be mechanically defined. Given the latter, the output of TELDAP, in turn, could be mechanically converted to actual test cases.

ACKNOWLEDGMENTS

I am indebted to my wife, Gertrude S. Elmendorf, for her willing ear and able editorial hand. I also acknowledge the contribution of the APL\ 360 system. The elegance of the APL language and the convenience of interactive computing were indispensable in developing and proving the TELDAP algorithm.

REFERENCES

1. D. B. Armstrong, “On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets”, IEEE Trans. on Electronic Computers, Vol. EC-15, No.1, February 1966, pp. 66-73.

2. R. A. Bender and E. L. Pottorff, “Basic Testing”, unpublished paper incorporated into Test Technology Notebook, Department D06, Poughkeepsie, June 11, 1970, 61 pp. [Editor’s Note: This was later published as IBM Technical Report TR 00.2108 “Basic Testing: A Data Flow Analysis Technique”, October 9, 1970.]

3. E. F. Codd, R. T. Burger and L. L. Lunde, “An Approach to Software Specification and Design”, TR 00.1588, May 31, 1967, 99 pp.

4. W. R. Elmendorf, “Controlling the Functional Testing of an Operating System”, TR 00.1729-1, January 10, 1969, 15 pp.

5. A. D. Falkoff and K. E. Iverson, APL\360: User's Manual, GH 20-0683, 1968.

6. S. M. G1assover, K. V. Hanford and C. B. Jones, “The Syntax Machine -- An Introduction”, TR 12.077, October 1968, 65 pp.

7. J. M. Galey, R. E. Norby and J. P. Roth, “Techniques for the Diagnosis of Switching Circuit Failures” IEEE Trans. on Communications and Electronics, Vol. 83, September 1964, pp. 509-514.

8. J. Hess, “Application of the Theory of Finite State Machines and Regular Expressions to APADS Program Testing System”, July 8, 1969, 10 pp. (Available from FSC Programming Laboratory, Gaithersburg, Maryland.)

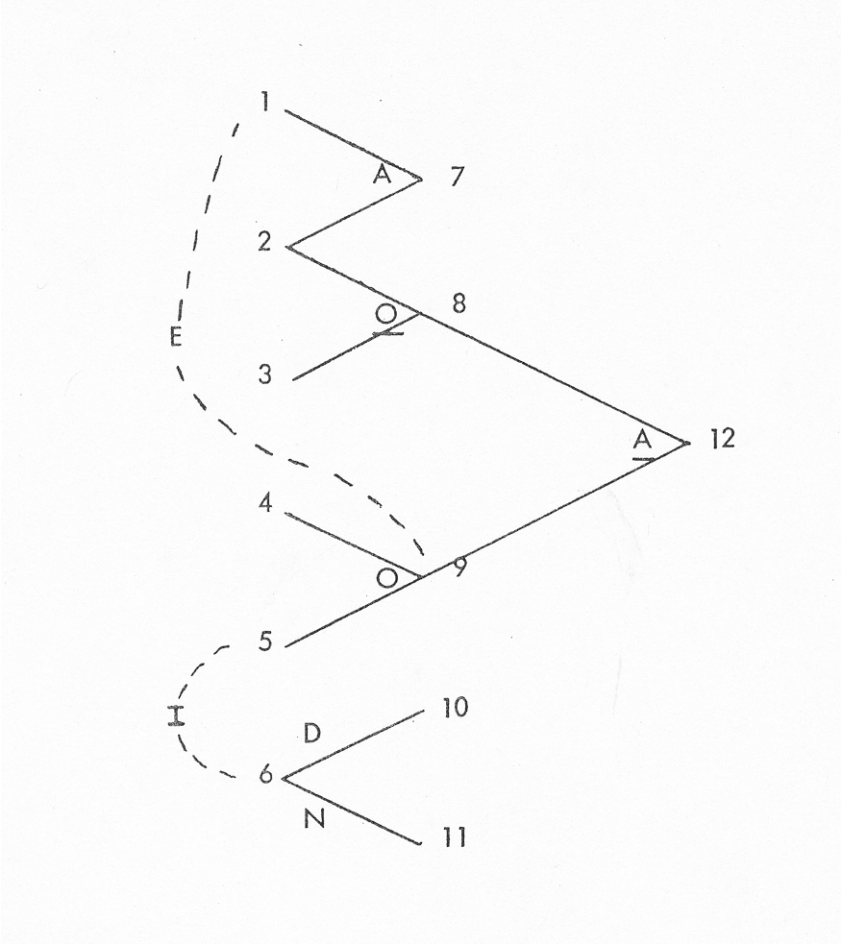
9. J. Hess, “Program Flow and Regular Expressions” memorandum to M. Dyer, September 12, 1969. (Available from FSC Programming Laboratory, Gaithersburg, Maryland.)

10. P. Herman and J. H. Pearson, “Technica1 Plan for Problem Determination and Program Reliability Improvement” unpublished report by San Jose RAS Programming Technology, December 12, 1969, 28 pp.

11. P. Lucas, P. Lauer and H. Stig1eitner, “Methods and Notation for the Formal Definition of Programming Languages”, TR 25.087, June 1968.

12. J. J. Maltby, H. T. Mehl and J. Safer, “Test Case Development System (TCDS)” TR 01.1152, March 20, 1970, 52 pp.

APPENDIX A. COMPREHENSIVE EXAMPLE



NODES 12
1 2 AND 7
2 3 NOR 8
4 5 OR 9
6 DIRECT 10
6 NOT 11
8 9 NAND 12
EXCLUSIVE 1 9
INCLUSIVE 5 6
INVOKABLE 1 2 3 4 5 6 9
OBSERVABLE 7 10 11 12

FUNCTIONAL VARIATIONS:

1. IF CAUSE(S) 1 ABSENT THEN EFFECT 7 ABSENT
2. IF CAUSE(S) 2 ABSENT THEN EFFECT 7 ABSENT
3. IF CAUSE(S) 1 2 PRESENT THEN EFFECT 7 PRESENT
4. IF CAUSE(S) 2 PRESENT THEN EFFECT 8 ABSENT
5. IF CAUSE(S) 3 PRESENT THEN EFFECT 8 ABSENT
6. IF CAUSE(S) 2 3 ABSENT THEN EFFECT 8 PRESENT
7. IF CAUSE(S) 4 PRESENT THEN EFFECT 9 PRESENT
8. IF CAUSE(S) 5 PRESENT THEN EFFECT 9 PRESENT
9. IF CAUSE(S) 4 5 ABSENT THEN EFFECT 9 ABSENT
10. IF CAUSE(S) 6 ABSENT THEN EFFECT 10 ABSENT
11. IF CAUSE(S) 6 PRESENT THEN EFFECT 10 PRESENT
12. IF CAUSE(S) 6 PRESENT THEN EFFECT 11 ABSENT
13. IF CAUSE(S) 6 ABSENT THEN EFFECT 11 PRESENT
14. IF CAUSE(S) 8 9 PRESENT THEN EFFECT 12 ABSENT
15. IF CAUSE(S) 8 ABSENT THEN EFFFCT 12 PRESENT
16. IF CAUSE(S) 9 ABSENT THEN EPFRCT 12 PRESENT

NEW FUNCTION TEST LIBRARY:

THIS LIBRARY CONTAINS 10 TESTS DOCUMENTED IN
CORRESPONDING ROWS OF THE FOLLOWING THREE MATRICES:

INVOKABLE CAUSE PATTERNS:

```
1  2  3  4  5  6  9
0  1  X X X X X
1  0  X 0  0  1  0
1  1  X 0  0  1  0
X 1  0  X X X X
X 0  1  X X X X
0  0  0  1  0  1  1
0  0  0  0  1  X 1
0  X X X  1  0 1
0  1  X X  X X 1
X 0  0  0  0  1  0
```

EACH ROW IS A TEST; EACH COLUMN IS AN INVOKABLE CAUSE
INTERPRET ELEMENTS AS FOLLOWS:

1: INVOKE THIS CAUSE IN THIS TES'T
0: SUPPRESS THIS CAUSE IN THIS TEST
X: INVOKE OR SUPPRESS THIS CAUSE IN THIS TEST

OBSERVABLE EFFECT PATTERNS:

```
      1 1 1
7 8 0 1 2

0 0 X X 1
0 X 1 0 1
1 0 1 0 1
X 0 X X 1
0 0 X X 1
0 1 1 0 0
0 1 X X 0
0 X 0 1 X
0 0 X X 1
0 1 1 0 1
```

EACH ROW IS A TEST; EACH COLUMN IS AN OBSERVABLE EFFECT
INTERPRET ELEMENTS AS FOLLOWS:

1: THIS EFFECT SHOULD OCCUR IN THIS TEST
0: THIS EFFECT SHOULD NOT OCCUR IN THIS TRST
X: THIS EFFECT MAY OR MAY NOT OCCUR IN THIS TEST

FUNCTIONAL MATRIX:

```
                      1 1 1 1 1 1 1
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
T
  T                     T T
    T                   T T
      T
        T
           T  T         T T   T
           T    T            T
                  T       T
T                                T
           T      T     TT         T
```

EACH ROW IS A TEST; EACH COLUMN IS A FUNCTIONAL VARIATION
INTERPRET ELEMENTS AS FOLLOWS:

T: THIS FAULTY VARIATION DETECTED BY THIS TEST
BLANK: THIS FAULTY VARIATION NOT DETECTED BY THIS TEST

OLD FUNCTION TEST LIBRARY:

THIS LIBRARY CONTAINS 6 TESTS DOCUMENTED IN CORRESPONDINC ROWS OF THE FOLLOWING THREE MATRICES:

INVOKABLE CAUSE PATTERNS:

1	2	3	4	5	6	9
0	1	0	X	1	0	1
1	0	1	0	0	1	0
1	1	X	0	0	1	0
0	0	0	1	0	1	1
0	0	0	0	1	X	1
X	0	0	0	0	1	0

EACH ROW IS A TEST; EACH COLUMN IS AN INVOKABLE CAUSE
INTERPRET ELEMENTS AS FOLLOWS:

- 1: INVOKE THIS CAUSE IN THIS TEST
- 0: SUPPRESS THIS CAUSE IN THIS TEST
- X: INVOKE OR SUPPRESS THIS CAUSE IN THIS TEST

OBSERVABLE EFFECT PATTERNS:

		1	1	1
7	8	0	1	2
0	0	0	1	1
0	0	1	0	1
1	0	1	0	1
0	1	1	0	0
0	1	X	X	0
0	1	1	0	1

EACH ROW IS A TEST; EACH COLUMN IS AN OBSERVABLE EFFECT
INTERPRET ELEMENTS AS FOLLOWS:

- 1: THIS EFFECT SHOULD OCCUR IN THIS TEST
- 0: THIS EFFECT SHOULD NOT OCCUR IN THIS TEST
- X: THIS EFFECT MAY OR MAY NOT OCCUR IN THIS TEST

FUNCTIONAL MATRIX:

```
                    1 1 1 1 1 1 1
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6

T     T           T     T   T
  T     T           T T
    T               T T
          T T       T T   T
          T   T             T
          T     T   T T       T
```

EACH ROW IS A TEST; EACH COLUMN IS A FUNCTIONAL VARIATION
INTERPRET ELEMENTS AS FOLLOWS:

T: THIS FAULTY VARIATION DETECTED BY THIS TEST

BLANK: THIS FAULTY VARIATION NOT DETECTED BY THIS TEST

APPENDIX B. TELDAP PROGRAM

[Editor's Note: This section had the APL code for the original program which began the automation of this process.]