# GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION
## SPONSORED PROJECT INITIATION

Date: May 3, 1979

Project Title: Experimental and Theoretical Research on Program Mutation

Project No: G-36-636

Project Director: Dr. R. A. DeMillo

Sponsor: Office of Naval Research; Code 613B:WRB; Arlington, VA 22217

Agreement Period: From 3/1/79 Until ~~2/28/80~~ ~~Aug 31, 1979~~ ~~8/31/83~~ ~~1/31/83~~ 1/31/84

Type Agreement: Contract No. N00014-79-C-0231 through GTRI.

Amount: $99,280.00

Reports Required: Progress Reports; Final Report

Sponsor Contact Person (s):

Technical Matters

Marvin Denicoff
Director, Information Systems
Program - Code 437
Mathematical & Information Sciences Div.
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Contractual Matters
(thru OCA)

Office of Naval Research
Resident Representative
Georgia Institute of Technology
Room 325, Hinman Research Building
Atlanta, Georgia 30332

Defense Priority Rating: DO-C9 under DMS Reg. 1

Assigned to: Information & Computer Science (School/Laboratory)

COPIES TO:

- Project Director
- Division Chief (EES)
- School/Laboratory Director
- Dean/Director—EES
- Accounting Office
- Procurement Office
- Security Coordinator (OCA)
- Reports Coordinator (OCA)
- Library, Technical Reports Section
- EES Information Office
- EES Reports & Procedures
- Project File (OCA)
- Project Code (GTRI)
- Other

CA-3 (3/76)

GEORGIA INSTITUTE OF TECHNOLOGY — OFFICE OF CONTRACT ADMINISTRATION

SPONSORED PROJECT TERMINATION/CLOSEOUT SHEET

Date May 11, 1984

Project No. G-36-636 — School/XXX ICS

Includes Subproject No.(s)

Project Director(s) Dr. R.A. DeMillo — GTRI / XXX

Sponsor Office of Naval Research

Title "Experimental and Theoretical Research on Program Mutation"

Effective Completion Date: 1/31/84 (Performance) 1/31/84 (Reports)

Grant/Contract Closeout Actions Remaining:

- [x] None
- [ ] Final Invoice or Final Fiscal Report
- [ ] Closing Documents
- [ ] Final Report of Inventions
- [ ] Govt. Property Inventory & Related Certificate
- [ ] Classified Material Certificate
- [ ] Other

Continues Project No. — Continued by Project No. G-36-661

COPIES TO:

Project Director
Research Administrative Network
Research Property Management
Accounting
Procurement/EES Supply Services
Research Security Services
Reports Coordinator (OCA)
Legal Services

Library
GTRI
Research Communications (2)
Project File
Other

Form OCA 60 1028

# Program Mutation: An Approach to Software Testing

Richard A. DeMillo
School of Information and Computer Science
Georgia Institute of Technology

# Program Mutation: An Approach to Software Testing

## Table of Contents

Program Mutation: An Approach to Software Testing

## Table of Contents

# Chapter 1

# Testing for Correctness

## Computability and Programming Systems

**Turing Machines.** We will assume familiarity with elementary computability theory. A Turing machine *decides* or *solves* a computational problem in the following way: when the machine is presented an input x, the machine eventually halts and either *accepts* or *rejects* the input. We say that a decision problem is *solvable* (or, equivalently, a predicate is *decidable*) if there is a Turing machine which accepts exactly those inputs which are solutions to the decision problem and rejects all others. Such a machine is said to be a *decision procedure*. A problem is said to be *unsolvable* if no decision procedure exists.

During its operation, a Turing machine carries out a number of basic operations (e.g., moving its read/write heads). The basic operations are called *steps*. If a Turing machine on input x carries out m basic operations and enters a halt state, the machine is said to have *halted after exactly m steps*.

We assume some canonical indexing of Turing machines. That is, an effective procedure whereby the ith Turing machine can be listed, for all $i \geq 0$. This indexing is fixed throughout.

The Kleene T-predicate is the predicate T(i,j,k) which is true exactly when the ith Turing machine (in the canonical listing of Turing machines), when given input j, halts in exactly k steps. The halting problem for Turing machines is the problem of deciding the truth of the the predicate $(\exists x)(T(i,j,x))$. The halting problem is unsolvable. The fundamental technique for showing that a problem is unsolvable will be to reduce the halting problem (or some other problem known to be unsolvable) to the problem in question. In general terms, such a proof involves showing how an aribtary instance of the halting problem can be transformed or reduced to an instance of the problem which is to be shown unsolvable in such a way that the Turing machine halts (or fails to halt) exactly when the transformed instance is a solution to the problem. The argument then proceeds as follows. If the problem is solvable, then the halting problem can be solved by applying the transformation to its instances and using the (assumed) decision procedure. Since this contradicts the unsolvability of the halting problem, the problem in question must also be unsolvable.

A Turing machine may also function as a transducer. That is, given an input x such that T(i,x,k), the ith Turing machine will write onto a designated portion of one of its tapes a value y. The function f determined by f(x) = y is said to be computed by the ith Turing machine. A function which is computed by some Turing machine is said to be computable.

An oracle Turing machine contains designated query states. In a query state, the machine submits a fixed value x to an oracle. If the oracle is for a function f, in one step the machine will respond to the query with $f(x)$. Notice that the oracle f need not be computable. The canonical indexing can be modified to include all oracle machines.

**Programming Systems.** Any model of effective computation is called a <u>programming system</u>. In a programming system, it is possible to construct representations for algorithms; each such representation is said to be a <u>program</u>. We identify a programming system **P** with the set of programs it defines. It is not necessary that a programming system be universal, only that all programs be effective. We will usually identify a programming system with the set of programs that can be written in the system. Thus examples of programming systems are the set of Markov algorithms, the set of straightline programs which compute polynomials of some fixed degree, the set of linear recursive programs schemes, and the set of syntactically correct APL programs.

We assume that each program in a programming system presented in a uniform way, (and, like Turing machines, can be uniformly indexed) and that each program is defined on an <u>input space</u>, **D**. The programming system defines a method of interpreting programs. If a program $P \in \mathbf{P}$ is started on an <u>input</u> $x \in \mathbf{D}$, the semantics of the programming system defines the manner in which values are assigned to input variables, machine states are altered and output is delivered. Since the input spaces of programming systems vary, we will assume that each input space **D** can be coded in a natural way into the nonnegative integers N.

Let **P** be a programming system. To each P in **P**, there corresponds a computable function $P^*$. The correspondence is as follows: for each $x \varepsilon$ **D** we determine the $n(x) \varepsilon$ N that encodes x, and execute P on x to obtain an output y; then $P^*(n(x))=n(y)$. We sometimes extend this notation to **P**: $\mathbf{P}^* = \{P^* | P\varepsilon\mathbf{P}\}$.

The <u>equivalence problem</u> for a programming system **P** is the following decision problem. Given programs P,Q ε **P** determine whether or not for all $x \varepsilon$ **D**, $P^*(x) = Q^*(x)$.

## The Programming Model

The testing theory described here differs from most theoretical studies in that we make some assumptions about how programs (in a programming system) are produced.

We assume that the intended behavior of a program is given by a function f — the <u>specification</u>. In practice, describing f is very difficult, perhaps as difficult as programming itself. For our purposes, however, we need only assume that <u>some</u> functional specification exists and it is that function which is to be implemented by the programmer.

The programming task itself resembles a root-finding procedure.

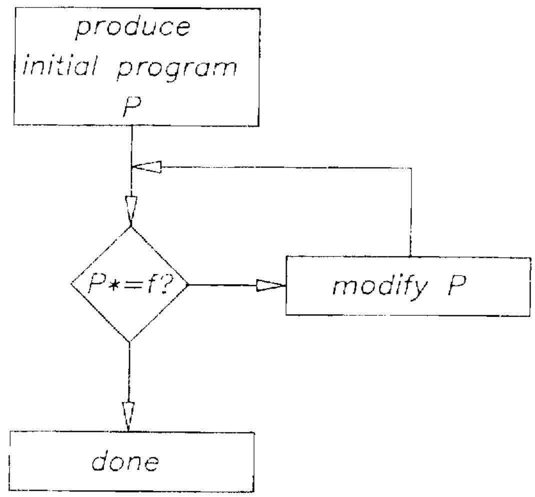



Figure 1.
The Iterative Programming Process

The initial program produced in Figure 1 corresponds to the initial guess of a root-finding procedure. During the initial iterations, the fact that the program at hand does not satisfy the specification will be obvious (e.g., the program is syntactically incorrect or has a run-time error). During later iterations, however, the P*=f test is carried out by direct comparison of the current version of P with f.

In the case that f is uniformly presented — for example, by a predicate calculus formula — the direct comparison may take the form of a <u>proof</u> of correctness. In the situation encountered most frequently in practice, however, f is not uniformly presented.

Rather, the programmer has available a number of <u>instances</u> of f of the form (x,f(x)). In this case, the determination of whether or not $P^* = f$ is made by observing a finite number of executions of P on instances of f. Since we want the theoretical development to be independent of any specific implementation of testing procedures, we will not distinguish these alternatives. Rather, we assume the existence of an <u>oracle</u> for f, i.e., a device for supplying instances of the form (x, f(x)) for finitely many $x \varepsilon D$.

A finite subset of D for which values of f are available is said to be a <u>test set</u> for P and f. Conceptually, f is an oracle for a procedure which executes P on an input x, queries f and checks $P^*(x) = f(x)$.

## Deductive and Inductive Inferences

We let P be an arbitrary but fixed programming system. We are interested in <u>testing</u> a program P with specification f during the interative process of producing a correct program.

**Definition:** P is <u>correct</u> with respect to a specification f if $P^*(D) = f(D)$. If P is correct with respect to f, the P is said to <u>compute</u> f.

A natural requirement for a test set that is useful in determining program correctness is that execution of the program on the test set should <u>demonstrate</u> the correctness of the program. Not every test set carries the same weight in demonstrating correctness. The testing process itself can be described by a rule of inference:

$$\frac{P^*(a_1)=f(a_1)\ \wedge P^*(a_2)=f(a_2)\ \wedge \ldots \wedge P\ (a_n)=f(a_n)\ \wedge \ldots}{P^*(D) = f(D)}$$

That is, from the *observations* $P^*(a_i) = f(a_i)$, the tester wishes to infer the generalization $\forall x \varepsilon D\ P^*(x) = f(x)$. Clearly, if the values $a_i$ run through all of D, the inference is deductively valid. But, in general, D is either infinite or large enough to make such procedure impractical.

Another way to view such an inference is in the context of an *experiment*. To establish the truth of the conclusion, the tester looks for *confirming instances* of the form $P^*(a)=f(a)$. If an experiment ever results in a value b such that $P^*(b) \neq f(b)$, then P is not correct, and the experiment has rejected the conclusion. On the other hand, the existence of a confirming instance does not guarantee correctness: there might be an undiscovered experiment that will show that P is incorrect. So the question arises: when does the tester *stop* experimenting and infer the correctness of P? In order to insure objective standards for testing P, these conditions should be stated in general terms as a *stopping rule*. We distinguish two forms of inference allowed by such rules. Suppose that a stopping rule R for a program P results in a set of values R(P) and experimental trials $P^*(x)=f(x)$ for $x \varepsilon R(P)$.

**Deductive Form:** From R(P) to infer that P is correct

**Inductive Form:** From $R(P)$ to infer that $P$ is correct with probability $\delta$.

Beyond the observation that the stopping rule should be useful in making either deductive or inductive inferences of this form, it is not at all clear what other properties stopping rules should have. Typical naive stopping rules (e.g., make voluminous tests, make tricky tests) have limited effectiveness. Useful rules are based on the following principle: the stopping rule should force the tester to produce a <u>strong</u> set of confirming instances. The notion of strong and weak confirming instances is particularly important in the context of testing program correctness since by simply compiling a finite table $\{(a_i, f(a_i)) \mid 0 \leq i \leq n\}$, a program can be easily modified to give correct output on a finite set of test cases.

To see the underlying problem in assessing the strength of confirming instances, consider the following thought experiment. By experimental observation, we are to determine whether or not

$$\forall x (A(x) \Rightarrow B(x)) \qquad (1)$$

is true. This entails finding confirming instances x such that $A(x)$ is true and checking to see that $B(x)$ also holds. But (1) is logically equivalent to

$$\forall x (\neg B(x) \Rightarrow \neg A(x)). \qquad (2)$$

Therefore, another experiment to check the validity of (1) might entail finding confirming instances y such that $B(y)$ <u>fails</u> and checking to see that $A(y)$ also fails. The problem is that strong confirming instances of (2) need not be strong confirming instances of (1). Suppose, for example, that (1) is the statement

"All ravens are black"

Then (2) states,

"All non-black objects are non-ravens."

Thus, while an experiment to verify (1) involves finding ravens and checking their colors, an experiment to verify (2) need not involve ravens at all. Strong confirming instances of (2) can be red shoes or gray walls, and such observations, while supporting a logically equivalent proposition, should provide no rational support for proposition (1).

To insure that the stopping rules which guide testing provide strong confirming instances of correctness, a number of possibilities have been suggested.

**Input Space Partitioning:** A path through a program P is a sequence of computations that correspond to a possible flow of control through the program. If a program contains loops, then differing numbers of iterations through loops give rise to different paths. It is possible to associate with every path $\pi$ a subset $D_\pi$ of D which causes that path to be executed. Thus, P* can be decomposed into a set of functions $P^*_\pi$, where $\pi$ runs through all paths in P, and the correctness of P can be determined by testing whether or not $P^*_\pi = f_\pi$, where $f_\pi$ represents the specification for the path $\pi$.

Consider a programming system P in which each program P satisfies the following condition: for each pair of paths $\pi_0$, $\pi_1$, $P^*_{\pi_0}(x) \neq P^*_{\pi_1}(x)$, for all $x \in D$. Suppose that we have obtained a stopping rule for each of the (possibly infinitely many) $P_\pi$ and that we can infer the correctness of each of them from the tests. Then we can use these tests to infer the correctness of programs in P if and only if $P^*_\pi(D_\pi) = f_\pi(D_\pi)$, for all paths $\pi$ implies that $P^* = f$. This latter condition is equivalent to requiring that domain of $f_\pi$ and $D_\pi$ be disjoint for all paths $\pi$, i.e., the path domains $D_\pi$ partition the domain D and the selection of points on which an incorrect program fails can be made randomly from the partitions.

Since the number of distinct paths in a program can be infinite the conditions given above are not particularly useful. On the other hand, it may be possible to choose a subset of all paths for consideration which is sensitive enough to guarantee that the inference can be made with a high degree of confidence. For example, the set of paths to be tested may involve only single iterations of loops and all non-looping paths.

**Random Testing:** Suppose that D is supplied with a probability distribution and that $p(x)$ is the probability that $P^*(x) \neq f(x)$, when x is chosen according to this distribution. Since p can be expected to converge to the failure rate when P is executed on $x \in D$ chosen according to the given distribution, we wish to derive a stopping rule which gives an indication of whether $p = 0$, after n tests. One way to derive an appropriate value of n is to calculate a quantity q based on the results of the tests so that q is greater than p with probability $1-\alpha$. If n tests are carried out and k instances x such that $P^*(x) \neq f(x)$ are observed, then q is the largest value of r such that

$$\sum_{i=0}^{n} \binom{n}{i} r^i (1-r)^{n-i} > \alpha.$$

Therefore, in a testing experiment, if no errors are observed

$$q = 1-\alpha^{1/n}.$$

The testing experiment, then, is to set the statistical limits on the confidence desired from the test (i.e., $1-\alpha$) and derive the appropriate value for n. Checking correctness on the random domain elements completes the test and allows the inference of correctness to be made.

If D is partitioned into m subsets $D_1,\ldots D_m$, then it may be possible to assess the probability $d_i$ that a random $x \in D$ is in $D_i$. For example, if the $D_i$ are path partitions and the paths correspond to functions that the program is to carry out, each function being selected with known distribution then $d_i$ is simply the probability that the ith function is selected. Similarly, if $p_i$ is the failure rate for the ith function determined by $D_i$, we have:

$$p = \sum_{i=1}^{m} d_i p_i.$$

Now, consider an experiment in which D is partitioned and for each $D_i$ $P^*_i(x) = f_i(x)$, where $f_i$ is the specification for the ith partition, for a random choice of x. Then regardless of the distribution of the $d_i$'s,

$$q \geq 1 - \alpha^{1/n}.$$

In this way a simple stopping rule can be used to give an inductive inference of correctness.

## Reliability of Test Data

The point of these techniques is to insure that the test set chosen allows the inference of correctness to be made with a high degree of confidence. However the test set is chosen, it should allow such an inference. Two versions of a stopping rule which are useful for such an inference are obvious generalizations of the rules given in the examples above.

**Deductive Stopping Rule:** Choose a set of test data so that correct performance on the test data implies correctness.

**Inductive Stopping Rule:** Choose a set of test data so that correct performance on the test data implies correctness with probability 1-p.

The first version provides a convenient characterization of test data which is strong enough to allow a valid inference of correctness.

**Definition:** A test set T is reliable for a program P and specification f if $P^*(T) = f(T)$ implies that P computes f.

Suppose that T is a reliable test set. If $P^*(T) = f(T)$, then by definition P is correct. On the other hand, if $P^* = f$, then $P^*(T) = f(T)$ for any subset of D. Thus, if a test set T is reliable for P and f, then $P^*(T) = f(T)$ if and only if P is correct. In essence, reliability of test data restates program correctness. For example, a proof that T is reliable for a correct program is by definition a proof of correctness. Unlike pure correctness proofs, finding a reliable test set for an incorrect program involves locating a program error, since $P^*$ and f must differ on at least one point of a reliable test set.

**Theorem 1:** For any P,f there is a reliable test set.

**Proof:** If P computes f then any test set will do. If P does not compute f, let $x \in$ **D** be any point for which $P^*(x) \neq f(x)$. Clearly $T = \{x\}$ is reliable. []

Given a program P to be tested, two related problems arise. On one hand we may be called upon to judge from available evidence whether or not P is correct. On the other hand, we may be called upon to produce evidence that is certain to convince such a judge. If the acceptance criteria is the existence of a reliable test set, the problems reduce to the following. Since P is correct exactly when it performs correctly on a reliable test set, a proof that T is reliable for P is a proof of correctness for P, provided only $P^*(T)=f(T)$. By the same token, a mechanical way of producing reliable test sets, implicitly provides mechanical proofs of correctness. Since every program has a reliable test set, procedures to prove that a test set is reliable and to generate reliable test sets are possible.

**Definition :** The <u>decision problem</u> for reliable test sets is to determine for program P, test set T, and specification f whether or not T is reliable for P and f.

**Definition :** Let G be a mapping from program-specification pairs to finite subsets of D. G is said to be a <u>reliable test strategy</u> if G(P,f) is reliable for P and f.

In referring to the decision problem for reliability and reliable test strategies we will not mention the underlying programming system or the specification when there is no danger of confusion. Thus, we will often refer to a test strategy for P, when the specification is clear from context.

A decision procedure for reliable test sets consists of a Turing machine with oracle f. P is encoded into the input alphabet of the machine (using, for example, the indexing function of oracle machines). When presented with P and an encoding of T, the procedure either accepts or rejects T.

**Theorem 2:** Assume that the decision problem for reliable test sets is solvable. Then there is a computable reliable test strategy.

**Proof:** Let $T_i \subseteq D$ consist of the first i elements of D under some effective ordering of D. By Theorem 1, there is a reliable test set for any (P,f), and any test containing it is also reliable. Thus, for some i, $T_i$ is reliable. The test strategy simply generates $T_0$, $T_1, \ldots$ at each stage testing to see whether or not the test set so far generated is reliable for (P,f). []

**Theorem 3:** If a programming system has a computable reliable test strategy, then the corresponding decision problem for reliable test sets is solvable.

**Proof:** Assume a reliable test strategy G. We decide whether or not T is reliable as follows. Given (P,f), we first produce a reliable test set G(P,f). By definition, if $P^*(G(P,f))=f(G(P,f))$, then P is correct and so <u>every</u> test set is reliable. The decision procedure thus should accept T as reliable. Suppose $P^*(G(P,f)) \neq f(G(P,f))$. Since P is not correct, T is reliable exactly when $P^*(T) \neq f(T)$. Since the process of checking $P^*(x)=f(x)$ for finitely many values of x can be carried out by a Turing machine which simulates P and queries an oracle for f, this procedure is a decision procedure.[]

Notice that the decision procedure above, does not really use any information about T when P is correct. This is simply a consequence of the fact that reliable test sets do not <u>demonstrate</u> correctness in any meaningful way. Indeed, if we have any independent proof that P is correct, then we can choose T as we please -- as a source of evidence to a third party who must be convinced of P's correctness this is not very satisfying. Furthermore, since the decision problem is equivalent by this argument to the decision problem for a powerful system of logic (e.g., the logic used to prove that P is correct), we would expect on intuitive grounds that the decision problem for reliability is, in general, unsolvable.

**Theorem 4:** There are classes of programs which have neither solvable decision problems nor computable test strategies.

**Proof:** Consider the following programming system $P = \{P_i | i \geq 0\}$. Each program $P_i$ is defined by the following specification:

$$P^*_i(x) = \begin{cases} 0, \text{ if } i=0 \\ 0, \text{ if } i>0, \text{ and } x \neq i \\ 1, \text{ if } i>0, \text{ and } x = i \end{cases}$$

It is easy to see that, since $P_i$ gives output 1 only when given its own index as input, $P^*_i = P^*_j$ exactly when $i=j$. It follows from this observation that the equivalence problem for **P** is solvable.

We claim that there is no computable test strategy for **P**. Suppose otherwise. A strategy G for $(P_0,f_0)$ queries $f_0$ a finite number of times and halts with some reliable T. Let i be an integer greater than any element of T and any element involved in a query for $f_0$. Then $G(P_0, f_i) = T$. Clearly T is not reliable for $(P_0,f_i))$, contradicting our choice of G.

By Theorem 2, the existence of a decision procedure for reliable test sets would also produce a computable test strategy, so the decision problem for reliable test sets is also unsolvable for **P**.[]

compiler certification) the expense of constructing a specification-sensitive device is justified by the number of programs which will be validated. Thus the <u>non-uniform</u> problem may be of interest.

**Definition** : Let the specification f be fixed and let P be a programming system. The <u>f-decision problem</u> for reliability in P is the problem of deciding, given $p \in P$ and test set T, whether or not T is reliable for P,f.

**Definition** : Let the specification f be fixed and let P be a programming system. An f-reliable test strategy is a mapping $G_f$ from P to finite subsets of D such that, for each $P \in P$, $G_f(P)$ is reliable for P and f.

The proof of the following theorem is nearly identical to the uniform case, and we omit it here.

**Theorem 6**: Let P be a programming system and let f be a specification. Then P has an f-decision procedure for reliability if and only if P has a reliable test strategy $G_f$.

Furthermore, just as in the uniform case, we can effectively obtain a test strategy from any f-decision procedure and conversely.

The equivalence problem for P also has the same relevance for the non-uniform problems, provided that we limit specifications to functions that are actually computed by some program in the programming system.

**Theorem 7:** If a programming system, P, has a decidable equivalence problem then its f-decision problem for reliability is solvable for each f $\varepsilon$ P*.

**Proof:** Let f be a specification in P*. Then some program $P_0$ $\varepsilon$ P computes f. Since we are dealing here with the non-uniform decision problem, no procedure for determining $P_0$ needs to be supplied. To decide whether T is reliable for P and f, we will use the decision procedure for equivalence: decide whether or not P = $P_0$. If so, then P is correct and T is therefore reliable. If P $\neq$ $P_0$, test P against specification f = $P_0$*. If P*(T) = f(T), then since T does not contain a point on which P fails, it is not reliable. On the other hand, if P*(T) $\neq$ f(T), then T is clearly reliable.[]

Not surprisingly (given Theorem 7), the ability to decide equivalence also gives enough power to compute a non-uniform test strategy. The proof of this fact follows closely constructions we have seen already, so we will not reproduce it here.

**Theorem 8:** If a programming system, P, has a decidable equivalence problem, then for each f $\varepsilon$ P*, there is a computable f-test strategy.

It might be hoped that restricting the decision or strategy problems to the non-uniform cases will make them easier. Unfortunately, reliability is such a strong property that, even in the non-uniform case, the decision (and hence the test strategy) problem is formally as hard as testing equivalence in the programming system.

**Theorem 9:** Let $f \in \mathbf{P^*}$ and suppose that P computes f. If some f-test strategy is computable, then the problem of deciding equivalence to P is solvable for all programs in **P**.

**Proof:** Suppose that $G_f$ is a reliable test strategy. Let $T = G_f(Q)$. If $Q^*(T) = P^*(T) = f(T)$, then, since T is reliable, $Q^* = f = P^*$. On the other hand, if $Q^*(T) \neq f(T)$, then $P \neq Q$. Therefore, to decide equivalence to P generate T and run the test for Q on T with specification $P^*=f$. The result of the test is the result of the decision procedure.[]

## Adequacy and its Measurement

Our first goal is to find a stopping rule which is as useful as reliability in inferring correctness, but which is also useful as evidence that a program is correct. Recall that the chief defect of reliability is that, if a program is correct, a reliable test set does not have to make any case at all for correctness. Our strategy will be to require that a test set provide an "explanation" of why the program is believed to be correct. For adequate test sets, this explanation simply states that the program is not incorrect and demonstrates this conclusion with test cases causing incorrect programs to fail but on which the original program does not fail.

**Definition:** Let f be a specification with domain of definition **D** for a program P (which may not be correct). A set of test data T is adequate for P with respect f if (a) $P^*(T) = f(T)$, and (b) for all programs Q such that $Q^*(\mathbf{D}) \neq f(\mathbf{D})$, $Q^*(T) \neq f(T)$.

In other words, T is adequate for P if P behaves correctly on T and all incorrect programs behave incorrectly on at least one element of T. Notice that the definition of adequacy incorporates correct execution on the test set as part of the definition while reliability does not. This makes comparisons between reliability and adequacy somewhat awkward. If T is adequate, then it is a simple consequence of the definitions that T is also reliable. On the other hand, suppose that P is correct. Then T = $\emptyset$ is reliable but not adequate. On the other hand, if P is incorrect, then it has no adequate test set, but it always has a reliable test set. Most of the theoretical developments based on adequacy can be are left intact if we use only part (b) of the definition. However, the goal of testing based on adequacy and related notions is to infer correctness. The usefulness of the process of deriving adequate test sets in revealing errors in incorrect programs is incorporated into experimenal implications of the theory.

**Theorem 10:** If T is adequate (for P), then T is reliable, but not conversely.

Recall from the previous section that reliable test sets always exist. Adequate test sets, on the other hand, must distinguish a program from a possibly infinite set of incorrect programs. Since this may require infinitely many test points, we cannot guarantee adequate test sets always exist even for correct programs.

**Theorem 11:** There are programming systems **P** such that for any program P $\varepsilon$ **P**, and any (finite) test set T, there is a function f such that $P^*(T)=f(T)$ but $P^*(x) \neq f(x)$ for all x $\varepsilon$ D-T.

**Proof:** Consider the set of straightline programs that compute polynomials. Let P be such a program and let $f=P^*$ be a polynomial of degree d. If T is any finite set, there is a program Q and polynomial $g=Q^*$ of degree $d' > d$ such that $f(T)=g(T)$ but f and g disagree on all points not in T. []

Notice that although T is reliable for P and f, it is reliable for neither (P,g) nor (Q,f), even though all agree on T.

**Corollary:** Let P be a set of straightline programs to evaluate polynomials. Then no program in P has an adequate test set for the specifications in $P^*$.

**Proof:** The proof of Theorem 11 gives an example of a program which for every finite test set agrees with an incorrect program. []

So far, we have been dealing exclusively with the deductive form of the inference problem. There is a probabilistic algorithm for the set of programs in Theorem 11. Denote by $\Pi(m,d)$ the class of m variable nonzero polynomials of degree d. Notice that the problem of determining whether or not $P^* = f$ can be turned into a problem about zeroes of polynomials by checking $P^*-f = 0$. Define $p(m,d,r)$ to be

$$\min \text{Prob}\{1 \leq x_1 \leq r,\ f(x_1,\ldots,x_m) \neq 0\}$$

where the minimum is taken over all $f \in \Pi(m,d)$. We derive a lower bound on $P = p(m,d,r)$ to get an upper bound 1-p on the error in selecting a random point from the m-cube. The procedure is then iterated t times to obtain an error probability of $(1-p)^t$. Since a polynomial of degree d has at most d roots, ignoring multiplicity, the largest probability of finding a root must be at least the probability of finding a root by random sampling in the interval $1 \leq x_1 \leq r$, and hence $p(1,d,r) \geq 1-d/r$. Now, consider some $f \in \Pi$. There are polynomials $\{g_i\}_{i \leq d}$ such that

$$f(x_1, \ldots x_m, y) = \sum_{i=0}^{d} g_i(x_1, \ldots, x_m) y^i.$$

Suppose that $g_k \in \Pi$. Then we have:

$$\mathrm{Prob}\{1 \leq x_i \leq r,\ f(x_1, \ldots, x_m, y) \neq 0\} \geq$$

$$\mathrm{Prob}\{g_k(x_1, \ldots, x_m) \neq 0,\ y \text{ not a root}\} \geq$$

$$p(m,d,r)(1-d/r).$$

Continuing inductively gives

$$p(m,d,r) \geq (1-d/r)^m,$$

and

$$\lim_{m \to \infty} (1-d/r)^m = \exp(-dm/r)$$

Thus, for large m and r=dm, we have $p(m,d,dm) \geq e^{-1}$. Therefore, with t evaluations of f for independent choices from the m-cube with sides r, a (finite) test set can be constructed which is adequate with probability $(1-e^{-1})^t$.

In the previous section, we examined the problem of deciding whether or not a test set is reliable and generating reliable test sets. We have the same interest in deciding test data adequacy and generating adequate test sets, if they exist. The definitions adapt readily to our purpose.

**Definition:** Let **P** be a programming system. The decision problem for adequacy in **P** is the problem of determining for a program $P \in \mathbf{P}$, a specification f and test set T, whether or not T is adequate for P,f.

**Theorem 12:** There is a programming system **P** such that the decision problem for adequacy in **P** is unsolvable.

**Proof:** We define a programming system $\mathbf{P} = \{P_i \mid i \geq 0\}$ as follows.

$$P^*_i(x) = \begin{cases} 0, & \text{if } i=0 \\ 1, & \text{if } i>0 \text{ and } T(i,i,x) \\ 0, & \text{if } i>0 \text{ and } \neg T(i,i,x) \end{cases}$$

Notice that for all values of i, $P^*_i(x)$ is defined for all values of x. $P^*_i$ is the constant zero if and only if the ith Turing machine fails to halt on all inputs, so the problem of deciding equivalence to $P_0$ is unsolvable.

We claim that an adequate test set exists for P and P* just in case $P \neq P_0$. Suppose $P^*(x) = 1$ and suppose that $Q^*(x) = 1$. Then Q and P both give the results of simulating some ith Turing machine for exactly x steps and must be equivalent. Thus {x} is an adequate test set. If $P^*$ is the constant zero function then there is no finite adequate test set since for every m there is a machine which halts on its index in more than m steps. Therefore, an adequate test set for $P_i$ exists if and only if $P^*_i$ is not identically zero, that is, $P_i$ is not equivalent to $P_0$. But equivalence to $P_0$ is undecidable, so the problem of deciding whether $P_i$ has an adequate test set must be unsolvable. []

Thus, two problems arise in connection with test data adequacy. First, adequate test sets need not exist. Second, as with reliability, adequacy is a <u>deductive</u> concept, and by virtue of this fact has an unsolvable decision problem. We would like to weaken the notion of adequacy slightly in order to remove both defects. The discussion following Theorem 11 provides some clues as to how this might be done. We would like a property of test sets that allows an inductive inference of correctness, preferably one that can be carried out with a fixed a priori probability of error. In practice, the probability of error may be determined by observations; in such situations, the inference of correctness will be a statistical inference whose strength depends on the strength of a fixed set of empirical observations.

**Definition:** Let f be a specification with domain D, let P be a program and let A be a set of programs (possibly depending on P). A set of test data T is adequate <u>relative</u> to A (with respect to f) if (a) $P^*(T) = f(T)$, and (b) for all programs $Q \in A$, if $Q^*(D) \neq f(D)$, then $Q^*(T) \neq f(T)$.

Thus, a set of test data is adequate for a program P relative to A if the data distinguishes P from all incorrect programs in A.

That adequacy relative to A is formally weaker than either adequacy or reliability is established by the following Theorem.

**Theorem 13:** If T is adequate for P relative to A, then either T is reliable or $P \notin A$.

**Proof:** Let T be adequate relative to A and suppose that T is not reliable. Then $P^*(D) \neq f(D)$. But for all $Q \in A$, if Q is not correct, then $Q^*(T) \neq f(T)$. Since $P^*(T) = f(T)$, P cannot be in A.[]

For example, A might represent a certain set of errors which are likely to be introduced into P. Then the existence of a test set T adequate relative to A demonstrates one of two things. Either P is correct (i.e., T is reliable) or P does not contain an A-type error. This property of relative adequacy fits nicely into inductive inferences. Suppose that $P \in A$ with probability $1-\delta$. Then if P has a test set T adequate relative to A, the probability that P subsequently fails is at most $\delta$ (if T is reliable then P fails with probability 0, and if P is not correct, then it is not in A, an event of probability $\delta$).

Therefore, if a set A can be found (or generated) which is extensive enough to insure that $\delta$ is small, the <u>inductive</u> inference can be made with a well-defined level of confidence.

Unlike adequacy, relative adequacy requires only "alternatives" in A be considered. If A has a particularly simple structure, then the problem of distinguishing P from A might be considerably easier than the problem of distinguishing P from <u>all</u> programs in the programming system. At this point, it is not at all clear what simple structure can be imposed upon A. However, two possibilities are likely candidates. The first is to require that A have a decidable equivalence problem. The second is to require that A be finite.

**Definition:** The <u>decision problem for relative adequacy</u> is the problem of determining for program P, subset A(P) of the programming system, and test set T, whether or not T is adequate relative to A(P).

**Definition:** Let G be a function that for program P, subset A(P) of the programming system, and specification f, defines $T = G(P, A(P), f) \subseteq D$. If all such T are adequate relative to A(P), then the function G is said to be an <u>adequate test strategy</u> (relative to A(P)).

If $A = \mathbf{P}$, then adequacy relative to A is simply adequacy. Therefore, it is possible that relatively adequate test sets do not exist, and a computable test strategy may be only a partial function.

**Theorem 14:** Assume that $A \subseteq \mathbf{P}$, that every program in **P** has an adequate (relative to A) test set and that there is a decision procedure for adequacy relative to A for **P**. Then there is a computable adequate test strategy for all programs in P.

**Proof:** As in the proof of Theorem 2, consider any decision procedure for relative adequacy. Given P, A and a specification, a test strategy simply enumerates subsets of D, deciding for each subset whether or not it it adequate relative to A. If a relatively adequate test set exists, the enumeration procedure will eventually discover a test set containing it, and output that set as the result of the strategy.[]

However, the converse does not hold

**Theorem 15:** The existence of a (total) computable adequate test strategy does not imply that the decision problem for adequacy is solvable.

**Proof:** Define a programming system $P = \{P_{ij} | 0 \leq i,j\}$ as follows. $P*_{i0}$ is the function that is i on input 0 and 0 otherwise. For all $j > 0$ let $P_{ij}$ compute the function $P*_{ij}$ defined below:

$$P*_{ij}(x) = \begin{cases} i, \text{ if } x = 0, \\ j, \text{ if } x=1, \\ 0, \text{ if } x=2, \text{ and } T(i,i,j), \\ 1, \text{ if } x=2 \text{ and } \neg T(i,i,j), \\ 0, \text{ if } x>2. \end{cases}$$

For each $P_{ij}$, let $A = A(P_{ij})$ be the set of programs $\{P_{ik}: k \geq 0\}$. Since {0,1} distinguishes any two programs in A, {0,1} is adequate relative to A. Hence the strategy that produces {0,1} is adequate and is clearly computable.

To show that adequacy relative to A is undecidable, notice that if the ith Turing machine halts in k steps, then $P^*_{ik}(2)=0$, and the test set {2} fails to distinguish $P_{i0}$ and $P_{ik}$. But $P^*_{i0}(1) \neq P^*_{ik}(1)$. If the ith Turing machine fails to halt on input i, then for all m, $P^*_{im}(2) = 1$ and {2} is adequate for $P_{i0}$. Suppose there is a decision procedure. Then the procedure announces that {2} is adequate relative to A for $P_{i0}$ iff the ith Turing machine fails to halt on input i. []

**Corollary:** There are programming systems with a decidable equivalence problem and for which every program has an adequate test set for which adequacy is not decidable.

**Proof:** Since the equivalence problem for the programming system P constructed above is decidable, the corollary follows immediately.[]

**Theorem 16:** There are programming systems with a decidable equivalence problem and for which adequate test sets exist for each program that do not have a computable adequate test strategy.

**Proof:** Let $P = \{P_{ij} | 0 \leq i,j\}$ be a programming system defined as follows. For each i,j, define

$$P^*_{ij}(x) = \begin{cases} i, & \text{if } x=0 \\ 1, & \text{if } 0<x\leq j \text{ and } T(i,i,x) \\ 0, & \text{otherwise.} \end{cases}$$

By construction, $P_{ij} = P_{km}$ exactly when i=k and $\neg T(i,i,n)$, where $\min(j,m) < n < \max(j,m)$. Clearly equivalence is decidable.

Choose $A(P_{ij}) = \{P_{im}: m \geq 0\}$. For given i, if the ith Turing machine fails to halt on input i, then all elements of A compute the same function, and so any nonempty test set is adequate for $P_{ij}$ relative to A. On the other hand, if $T(i,i,m)$, then $\{0,m\}$ is adequate. Thus, each program, P, has an adequate test set relative to A(P). Assume that a computable strategy, $G_A$, exists, and consider $G_A(P_{ij})$. The ith Turing machine halts on input i iff it halts at the mth step, for some m in $G_A(P_{ij})$. Since test sets are finite, this is impossible. []

Therefore, there are some very bad choices for A, indeed. Even assuming that A has a decidable equivalence problem does not improve the situation much. We will now examine the effects of requiring only that A be finite.

**Definition:** Let **P** be a programming system. For each program P, let $\mu(P)$ be a finite subset of **P**. Assume further that $\mu$ is computable in the sense that there is an effective procedure that lists $\mu(P)$ for all P. $\mu(P)$ is said to be a set of <u>mutants</u> of P.

**Theorem 17:** Every correct program has a test set adequate relative to $\mu(P)$.

**Proof:** There are only finitely many programs Q in $\mu(P)$ and each such Q is either correct or not. If $f(x) = P^*(x) \neq Q(x)$, add x to the test set. Only finitely many points need be added to obtain an adequate (for $\mu(P)$) test set. []

**Definition:** The $\mu$ equivalence problem is that of deciding whether or not $Q \varepsilon \mu(P)$ and $P = Q$.

**Theorem 18:** The following statements are equivalent.

(a) the $\mu(P)$-adequate decision problem is solvable.

(b) there is a computable $\mu(P)$ test strategy.

(c) the $\mu$ equivalence problem is decidable.

**Proof:** If there is a $\mu(P)$ decision procedure, then a computable $\mu(P)$ test strategy may be constructed as in the proof of Theorem 2. Thus, (a) implies (b).

To show that (b) implies (c) assume a computabe strategy. Given programs P,Q decide $\mu$-equivalence as follows. Compute $\mu(P)$ and check $Q \varepsilon \mu(P)$, and reject if not. Otherwise, generate a test set which is adequate relative to $\mu(P)$ and check equality of $P^*$ and $Q^*$ on this set. By the definition of adequacy, equality on the test set implies equality over D.

Suppose that we are given a decision procedure for $\mu(P)$ equivalence, and we are to decide whether a test set T is $\mu(P)$-adequate for specification f. Assume that $P^*(T)=f(T)$. First, construct the set $\mu(P)$ and determine those $Q \varepsilon \mu(P)$ which are not equivalent to P. This procedure is effective. For each such $Q \neq P$, we search for some $x \varepsilon T$ such that $P^*(x) \neq Q^*(x)$. Obviously, T is adequate if and only if each such search is successful. Therefore, (c) implies (a).[]

Although there is an equivalence between the decision problems for $\mu(P)$ adequacy, equivalence and test strategies, the finiteness of $\mu(P)$ alone is not sufficient to guarantee that any of these problems are solvable.

**Theorem 19:** There are programming systems **P** and functions $\mu$ so that none of (a)-(c) in the statement of Theorem 18 are true.

**Proof:** Let **P** be as constructed in the proof of Theorem 12, and let $\mu(P) = \{P_0, P\}$ for all $P \in \mathbf{P}$. Then {0} is adequate for $P_i$ iff the ith Turing machine on input i does not halt. Since the decision problem for adequacy is unsolvable, Theorem 18 can be used to complete the proof. []

In order for $\mu(P)$-adequacy to be useful in practice, we evidently have to exercise some care in defining $\mu$, insuring that either the appropriate decision problems are easily decidable, or that heuristics are available.

A key aspect of $\mu(P)$-adequacy is that it admits <u>measurement</u> of how close a given test set is to being adequate. This is a relaxation of the decision problem for adequacy which is frequently encountered in testing situations. Since $\mu(P)$-adequacy may itself be a (statistically) strong predictor of program correctness, it may not be cost effective to develop a test set which is $\mu$-adequate. Rather, the inference of correctness may be made on much more slender foundations: the test set is "almost" adequate. We will consider the definition of such a measure here. In later chapters we will consider the evidence for its effectiveness as a stopping rule.

Let $\mu_E(P)$ be the set of those programs in $\mu(P)$ which are functionally equivalent to P; that is, $Q \varepsilon \mu_E(P)$ if $P^*(D) = Q^*(D)$. For a set of test data T, we define $\Delta(P,T)$ to be the set of programs $Q \varepsilon \mu(P)$ which disagree with P on at least one point in T. We will confuse the size of a set with its cardinality; in particular, $\mu(P)$ will be used to denote $|\mu(P)|$. Then the <u>mutation score</u> of T is the fraction of the nonequivalent elements of $\mu(P)$ which differ from P on one or more points in T:

**Definition:** The mutation score of T for P is defined to be

$$m(P,T) = \Delta(P,T)/\mu(P)-\mu_E(P).$$

Notice that once $\mu(P)$ is fixed, $\mu_E(P)$ and $\Delta(P,T)$ are determined by the semantics of the programming system. We want m to be a measurement of test data <u>quality</u>. That is, the function m should be useful in a stopping rule for inductive inferences of correctness: it should be possible to choose a function $\mu$ so that

(a) $\mu(P)$ is relatively easy to compute, and

(b) $m(P,T)$ approaches one as our confidence in the correctness of P increases by virtue of P's correct execution on T.

It is an easy observation that $m(P,T)$ is a direct measurement of how close the test set T is to being adequate for P relative to $A = \mu(P)$.

**Theorem 20:** Assume that $\mu(P)$ contains a correct program. Then $P^*(T) = f(T)$ and $m(P,T) = 1$ implies that T is adequate for P relative to $\mu(P)$.

**Proof:** Assume that $\mu(P)$ contains a correct program Q, and suppose that $P^*(T) = f(T)$ and $m(P,T) = 1$. If P is correct, then for any program R, $R \neq P$ iff $R \neq f$. If $R \in \mu(P)$ and $R \neq f = P$ and if $m(P,T) = 1$, then $R^*(T) \neq f(T)$. We claim that P cannot be incorrect, for suppose otherwise. Since $\mu(P)$ contains a correct program Q, $m(P,T)$ cannot be 1 unless $P^*(T) \neq Q^*(T) = f(T)$, a contradiction.[]

The assumption that $\mu(P)$ contains a correct program is called the <u>Competent Programmer Assumption</u>. The competent programmer assumption is a limiting empirical hypothesis. In a previous section (see Figure 1) we defined the programming model by analogy with a root finding procedure in which the process of creating and debugging a program can be stated

$$P_f = \text{(valid representation of program correct for f)}.$$

The program playing the role of the iterative in this process can be expected to change less and less as the programming process continues. When the program is "close" to a correct program, the process stops. Thus, a program to be evaluated by any of the techniques described above is not a <u>random</u> response to a specification: if it has been produced by a <u>competent</u> programmer, it has already been subjected to the iterative programming process. Therefore if $\mu(P)$ represents those programs which are <u>close</u> (in the sense of root-finding) to a correct program, with high probability, P will either be correct or within a small neighborhood of a correct program. Our goal in subsequent chapters will be to define $\mu(P)$ so that this assumption is useful in practice.

Theorem 20 can be restated in another form which is often more useful. The specific function $\mu$ we will deal with later behaves in a "reversible" manner; that is, $P \in \mu(Q)$ if and only if $Q \in \mu(P)$. Theorem 21 follows by an argument similar to the one above.

**Theorem 21:** If $P^*(T) = f(T)$ and $m(P,T) = 1$, then either T is correct or for all correct programs Q, $P \notin \mu(Q)$.

Therefore, by analogy to Theorem 13, we have a measurement of test quality which either accurately reflects the reliability of the test data or requires the violation of a specific empirical hypothesis.

## Bibliographic Notes

There are several good references on elementary computability theory. Perhaps the most accessible of these are the classic texts by Davis [Davis, 1958] and Minsky [Minsky, 1967]. The notions dealing with inductive and deductive inferences are implicit in most systematic treatments of logical and mathematical matters, and nearly any logic text provides the basic definitions. The relationship of deductive techniques to program correctness is discussed critically in [DeMillo, 1979]. Budd's dissertation [Budd, 1980] gives a good overview of the importance of inductive reasoning in program testing and uses the example of the black ravens.

Many additional sources of information concerning alternative test techniques can be found in the literature. Input space partitioning methods are discussed by Howden [Howden, 1976] and White, Chandrasekaran and Cohen [White, 1978]. The probabilistic algorithm for testing zeroes of polynomials is due to DeMillo and Lipton [DeMillo, 1978]. The algorithm is related to a problem in algebraic program testing [Howden, 1976].

Test data reliability was defined by Howden [Howden, 1976] and similar concepts have been given formal treatment by a number of authors. The paper [Goodenough, 1975] also treats the notion of reliable test set generation. Test set adequacy was formulated by DeMillo, Lipton and Sayward in [DeMillo, 1978a] and has been refined in a series of papers [Acree, 1979], [Budd, 1980a], DeMillo, 1979a]. The relationship between adequacy and mutant programs was developed concurrently and this development can be traced in [DeMillo, 1978a], [Acree, 1979], [Budd, 1980], [DeMillo, 1979a], [Acree, 1980]. Related concepts have appeared in [Foster, 1978], [Hamlet, 1978], [Howden, 1982], and [Brooks, 1980]. The relationship between program equivalence, test generation and recognition problems was worked out in a paper by Budd and Angluin [Budd, 1980].

# Chapter 2

# Errors and Mutations

## The Competetent Programmer Assumption

Let us recall the following definitions from Chapter 1. If $\mu$ is a mapping which associates a set of programs with a given program P, $\mu_E(P) \subseteq \mu(P)$ is the set of programs in $\mu(P)$ which are functionally equivalent to P, and if for a given test set T, $\Delta(P,T)$ consists of those programs in $\mu(P)$ which disagree with P on at least one point in T, then the measure

$$m(P,T) = \Delta(P,T)/\mu(P)-\mu_E(P)$$

can be defined. Theorem 1.21 guarantees that if P executes correctly on the test set T and m(P,T)=1, then either P is correct or P does not belong to $\mu(Q)$ for any correct program Q.

For a given program P, the set $\mu(P)$ is called a set of <u>mutants</u> of P. Thus, if every program P is a mutant of some correct program, calculation of the measure m(P,T) can be used to infer correctness.

The assumption that any program being tested is a mutant of a correct program is called the <u>Competent Programmer Assumption</u>. The Competetent Programmer Assumption formalizes an observation of human activity. In this case, the observation is that programmers do not create programs at <u>random</u>. Rather, programs that are written by experienced programmers, are written in response to formal or informal understandings of what the program is intended to do. Thus, in response to specifications for a payroll system, a competent programmer will produce a program that is very much like a correct payroll system. The program produced may be incorrect, inefficient or sloppy, but in the final analysis, it will be more like a correct payroll system than a compiler. The competent programmer assumption asserts that programmers create programs that are close to being correct. During the iterative programming process, competent programmers constantly whittle away the distance between what their programs look like now and what they are intended to look like.

Suppose that the task at hand is to design a Fortran program to compute the (Euclidean) magnitude of an N-dimensional vector X in a Cartesian coordinate system with fixed origin. Then the subroutine P1 below certainly could have been produced by a competent programmer.

```
  SUBROUTINE P1(X,MAG)
  MAG = 1
  DO 1 I = 1,N
  MAG = MAG+X(I)**2
1 MAG = SQRT(MAG)
  RETURN
  END.
```

We would question the competence of a programmer who produced subroutine P2:

```
  SUBROUTINE P2(X,MAG)
  MAG = X(1)
  DO 1 I = 1,N
1 MAG = MAX(X(I), MAG)
  RETURN
  END.
```

There is no reasonable sense in which P2 is a "buggy" version of the program asked for. P1 can easily be debugged, but P2 is not even a program of the same kind — it is so radically incorrect that its incorrectness can be discovered without testing it!

The competent programmer assumption states that a program is assumed to be either correct or a mutant of a correct program. For example, in the problem of computing magnitudes of N-vectors, subroutine P1 is a mutant of the correct P below.

```
  SUBROUTINE P(X,MAG)
  MAG = 0.0
  DO 1 I = 1,N
1 MAG = MAG+X(I)**2
  MAG = SQRT(MAG)
  RETURN
  END
```

Subroutine P2, on the other hand, is not a mutant of P.

The notion of closeness is summarized by the function $\mu$. Informally speaking, the set of mutants of a program P should reflect the possible errors that might have been made in the creation of P by a competent programmer. If a general concept of error can be derived in such a way that the Competent Programmer Hypothesis can be shown to hold with probability $1-\delta$ then the calculation of $m(P,T)=1$ allows an inference of correctness with the same level of confidence.

The classification of programming errors is not a well understood process. However, it appears that there are at least four mechanisms responsible software errors.

1. failure to satisfy specifications due to an implementation error,

2. failure to satisfy a requirement,

3. failure to write specifications that correctly represent a design, and

4. failure to understand a requirement.

The problems surrounding requirements and specification testing and evaluation are beyond the scope of this book and are probably not within the domain of correctness testing. The mechanisms referred to in (1) and (2), however, are always reflected in specific program errors: either a program carries out an action that it should not, fails to carry out a necessary action, or carries out an action improperly. This suggests that errors resulting from (1) and (2) are reflected in programs as missing control paths, inappropriate path selection, and inappropriate or missing actions.

In order to satisfy the Competent Programmer Assumption, carry out the following conceptual experiment. We observe a community of programmers and classify the errors they make into categories

$$E_1, E_2, \ldots, E_k.$$

We are free to observe the programmers for as long as we wish and make whatever specialized assumptions we wish about the programming task they will be called upon to perform. It is, in principle, possible to gain whatever degree of confidence we desire that among the k classifications we have encountered the errors most likely to be made by this particular group of programmers. Given a program P to test in this setting, we must derive a relatively adequate set of test data, T, for P. If P is incorrect, we will never be able to find an adequate set; indeed, the point of testing P is to find a set of test data that calls attention to the fact that P is incorrect. If P is correct, however, adequate T should at least convince us that P does not contain the errors most likely to be made.

Let

$$\mu(P) = \{P_1, P_2, \ldots, P_m\}$$

differ from P only in each containing a single error chosen from one of the error categories. Then an adequate set of test data T should at least provide the following assurance. For each $P_j$ which is not equivalent to P, $P^*(D) \neq P_j^*(D)$. In other words for each of the most likely errors, it should be possible to show that P does not contain that specific error. This experiment is specialized to the original group of programmers whose errors we observed and recorded. To attempt such an experiment for <u>all</u> programmers is surely hopeless, unless we can be assured that typical programmers tend to make the same, classifiable errors.

## Error Classification

The strength of the technique described above rests on our ability to assess the errors that programmers are most likely to make. Rather than speculate on the sources of errors, it is probably more fruitful to <u>examine</u> the errors that programmers actually do make.

A number of studies of programmer errors have been conducted over the years. These studies have been carried out using a variety of programs, error classification schemes, and methods for detecting errors. While several researchers have pointed out methodological flaws in the reporting, classification, and documenting of program errors, at least 46 independent, large-scale error data gathering efforts have been carried out and reported. For the most part, problems arising from error classification arise when data gatherers try to interpret the errors arising from the mechanisms (3) and (4) described above. However, the data on errors arising from mechanisms (1) and (2) show remarkable consistency.

The following data is based on E.A. Young's analysis of 69 programs and a total of 1,258 errors in several languages.

| Error Type | No. of Errors | Rel. Freq. |
|---|---|---|
| Job Ident. | 1 | 0.00 |
| Exec. Request | 1 | 0.00 |
| External I/O | 0 | 0.00 |
| Other System | 0 | 0.00 |
| Subrout. Ident. | 3 | 0.00 |
| Allocation | 189 | 0.15 |
| Label | 20 | 0.02 |
| Computation | 343 | 0.27 |
| Non-comput. | 2 | 0.00 |
| Iteration | 117 | 0.09 |
| GO TO | 13 | 0.01 |
| Conditional | 59 | 0.05 |
| I/O Format | 71 | 0.06 |
| Other I/O | 91 | 0.07 |
| System Call | 35 | 0.03 |
| Subrout. Call | 22 | 0.02 |
| Par/Sub List | 62 | 0.05 |
| Subrout. Term. | 7 | 0.01 |
| Other/Multiple | 72 | 0.06 |
| Data | 27 | 0.02 |
| Vert. Delim. | 54 | 0.04 |
| None | 69 | 0.05 |
| | 1258 | 1.00 |

Table 1. E. A. Young's Error Data

What is is striking about this data is the relatively small contribution of sophisticated error conditions. Errors such as operating system interface errors, incorrect job identification, and erroneous external I/O assignments accounted for only negligible quantities of the observed errors. It might be the case, however, that the significant contributors to the major error categories were themselves complicated errors. We will describe in a little more detail the nature of the errors which Youngs discovered.

**Allocation:** These included errors in declaring shapes and sizes of data structures as well as errors in allocating and deallocating local storage for named data objects. These errors accounted for 15% of the total. Almost all of them appeared in Algol, Cobol, or PL/I programs.

**Computation:** These errors occurred within assignment statements and comprised 27% of the observed errors. Almost half of them were caused by the use of a wrong variable or other data object. Wrong variable usage constituted the highest percentage. A large number of errors in this class stemmed from failures to initialize variables properly.

**Iteration:** Iteration sequence difficulties were semantic in nature (111 of 117). A typical example of such an error is an error in the number of loop iterationsresulting from a confution of DO and FOR loop semantics. Other examples include errors in loop scope and nonterminating loops. These errors accounted for 9% of the total.

**I/O:** 13% of the errors were due to I/O deficiencies, although most of these were syntactic in nature. Other common errors include the reading or writing of incorrect variables.

**Parameter/Subscript List:** Although 5% of the total were attributed to these errors, more than than sixty percent of the errors in this category were due to mismatching formal and actual parameters.

**Conditional Branch/Execution:** Most of these errors resulted from testing incorrect variables or using the wrong test in a conditional expression. These errors accounted for 5% of the total.

A second study was conducted by T. A. Thayer and his colleagues at TRW's Space Systems and Defense Group. The TRW classification broadly groups errors into twenty categories. We will concentrate on 4 categories which altogether account for 80% of the errors recorded in a study of two large-scale software development projects. The following distribution of reported errors is shown in Table 2.

| | Percent of Total Errors | |
|---|---|---|
| Major Error Categories | Project A | Project B |
| Computational | 9.0 | 1.7 |
| Logic | 26.0 | 34.5 |
| Data Handling | 34.6 | 36.1 |
| Interface | 17.0 | 22.5 |
| Data Definition | 0.8 | 3.0 |
| All Others | 12.6 | 2.2 |

Table 2. TRW Error Data

**Computational Errors:** These were errors introduced into arithmetic computations (the classification is insensitive to the nature of the computation; the computation could be the actual calculation of a physically interpretable quantity or merely a bookkeeping calculation of no significance outside the program). The calculations themselves occurred in assignment statements. The errors which make up this category include the incorrect use of an operand in an equation, the incorrect use of parentheses, an error in sign convention, an error in units or data conversion, the production of over/under flow in a computation, the application of an incorrect or inaccurate equation, and the loss of precision due to mixed mode arithmetic, and missing computations.

**Logic Errors:** The TRW classification scheme is vague about exactly what constitutes a logic error. Indeed, the assignment of specific errors to the logic category varied with the data gathering procedures. However, the studies published using this classification all seem to point toward errors which somehow affect logical decisions in the source code, even though the error under consideration may, in fact, be the result of failing to include a decision. Thus errors in this category included missing logic or condition tests. Logic errors also resulted from a lack of code to perform logical functions. Other errors which were classified as logical errors related to code written to carry out some particularly troublesome function (e.g., checking the settings of switches), or code which was erroneous due to misunderstandings of requirements or specifications. These resulted in incorrect operands in logical expressions, logic activities coded out of sequence, checking wrong variables, errors in the scope of loops, errors in the number of loop iterations, and duplicated logic.

**Data Handling Errors:** These errors included errors in input and output operations and errors in internal data handling. Typical data input errors included errors due to reading invalid input from the correct data file and reading from incorrect files. Also of significance were errors due to incorrect input formats and end of file processing. Internal data handling errors included errors in initializing data storage areas, using variable before they had been properly set, incorrect type usage, and subscripting errors. Finally, the data output errors mirrored the input errors. Errors such as garbled output or output not matching requirements were also considered. In addition, data definition errors such as errors in dimensions, referencing out of array bounds and pointer handling were also be classified as data handling errors.

**Interface Errors:** These errors roughly correspond to those that were introduced in the process of integrating program units or modules. These included calls to incorrect subroutines, misplaced subroutine calls, and errors in parameter passing during an invocation of a module.

The remaining errors considered in the TRW studies involved errors which were introduced and detected at other phases of the software lifecycle. They included operator/user errors, documentation errors, errors in interfacing to systems software, and requirements errors. In contrast, the remaining errors tended to be fairly complex and difficult to associate with specific program characteristics.

## Mutant Operators

Practice may dictate so many error types that the calculation of mutation scores becomes intractable. By concentrating only on "simple" mutants of P the technique becomes manageable. For example, in the case of computing magnitudes of vectors, P1 is not a simple mutant of P, but M1 and M2 are simple:

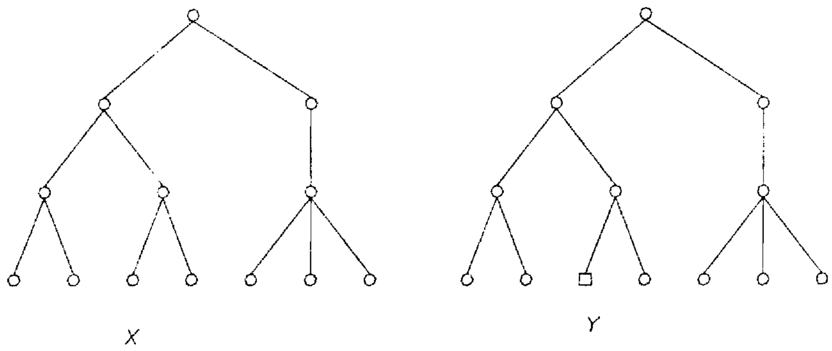
```
      SUBROUTINE M1(X,MAG)
      MAG = 1
      DO 1 I=1,N
    1 MAG = MAG+X(I)**2
      MAG = SQRT(MAG)
      RETURN
      END
```

```
      SUBROUTINE M2(X,MAG)
      MAG = 0.0
      SO 1 I=1,N
      MAG = MAG+X(I)**2
    1 MAG = SQRT(MAG)
      RETURN
      END.
```

The mutants we will consider arise from the single application of a mutant <u>operator</u>, a simple syntactic or semantic program transformation such as changing a particular instance of a relational operator to one of the remaining operators or changing the target of an unconditional transfer to another labelled target. A problem that arises immediately is that this is apparently a violation of the Competent Programmer Assumption. While error classification data indicates that programmer errors fall into a small number of identifiable categories, there is little to suggest that programmers make errors one at a time. Thus, while concentrating on simple errors may allow a tester to derive adequate test sets relative to a small class of errors, the data may not be adequate relative to a set of errors that are most likely to occur in practice. In fact, there is little lost in restricting mutants to those which can be defined by simple errors. As we will discuss below there is an observable <u>coupling</u> of simple and complex errors so that test data that causes all nonequivalent simple mutants to die is so sensitive that <u>likely</u> complex mutants also die. The coupling of simple and complex errors implies that if P is correct for an adequate test T while M1 and M2 disagree with P, then P1 must also disagree with P on T.

A set of mutants $\mu(P)$ is defined by a set of mutant operators that model a set of errors according to the Competent Programmer assumption. That is, for each error category $E_i$ there is a set of programs $\mu_i(P)$ which corresponds to the errors defined by $E_i$. There is no single correct set of mutant operators -- the Competent Programmer Hypothesis is specialized to a given community of programmers. In practice, however, it is usually only necessary to consider a fixed set of mutant operators which are derived from error data such as the data presented above.

One way to view mutation operators is a mapping between representations of source programs (see Chapter 4 for details on implementation strategies). Let the tree $T_1$ represent some program P, parsed into a tree-structured form as shown in Figure 1(a). Then a mutation operator when applied to $T_1$ produces a new tree $T_2$ by modifying a single leaf t of $T_1$ as shown in Figure 1(b).




(a)
Tree $T_1$

(b)
Tree $T_2$

Figure 1.
Mutation by Modifying a Leaf
of a Parse Tree

The tree $T_2$ remains a valid internal representation of <u>some</u> mutant program of P. In practice, not all of the mutant operators fit exactly into this model, but it is nevertheless a helpful organizing principle.

The result of applying such an operator is a <u>1-order</u> or <u>simple</u> mutant of the original program. <u>2-order</u> mutants are the result of two applications of (not necessarily the same) mutant operators. Continuing inductively, the notion of a <u>k-order</u> mutant can be defined for any $k \geq 1$. Since the result of applying a mutant operator always results in a syntactically correct program, the number of k-order mutants is given by $\xi_n^k$, where

$$\xi_n = \max\{\mu(P) \mid \text{size}(P) = n\}$$

and size(P) is any convenient size measure (see Chapter 5).

Unless specified otherwise, the term mutant will apply to simple mutants, and the set of mutants of P, $\mu(P)$, will be defined in terms of (simple) mutant operators. When we want to distinguish $\mu(P)$ from k-order mutants for some $k \geq 2$, we will use $\phi(P)$ for the set of complex mutants.

We now define a set of mutant operators which will form a basis for much of the rest of this book. These operators are mainly language independent with appropriate adaptation can be used as a core of mutant operators for machine implementation. Furthermore, the operators introduced below are designed to model error categories as described above. The effectiveness of the operators in modelling and detecting errors will be taken up in more detail in later chapters.

Mutant operators can be classified according to whether they affect operands, operators, or statements as a whole.

**Operand Mutants:** Mutations which affect operands alter the data objects of the program. For simplicity, we assume that there are three kinds of data objects: constants, scalar variables, and arrays. Thus there are nine mutant operators which replace a variable x with each distinct occurrence of y, where x and y range over all constants, scalar variable and array references in the program being tested.

In addition to these operators, there is an operator which alters the values of constants appearing in the program. The following table defines the alterations according to the type of the object to which the operators is applied.

| | |
|---|---|
| Integers | $\pm$ 1 |
| Nonzero reals | $\pm$ 10% |
| Real zero | $\pm$ .01 |
| Boolean | complement |
| Strings | replace first character by adjacent character in collating sequence |

Table 3. Data Mutations

A third type of operand mutation replaces array names in each occurrence of an array expression with all other array names of the same dimensionality. In specializing these operators to particular languages, additional operators which account for language dependent features may be needed to augment this list (cf. data mutations for Cobol).

**Operator Mutations:** Arithmetic operator mutations are formed by replacing each arithmetical operator with an operator chosen from the set $\{+,-,/, *,**,\lceil,\rceil\}$, where $\lceil$ and $\rceil$ are operators described below.

Relational operators are mutated by replacing each relational operator with an operator chosen from the set $\{<,\leq, =, \neq, \geq,>$, **trueop**, **falseop**$\}$, where **trueop** and **falseop** are the operators described below. Similarly boolean operator mutations are formed by replacing each boolean operator with an operator chosen from the set $\{\vee, \wedge$, **leftop**, **righttop**, **trueop**, **falseop**$\}$.

Each unary operator may be removed by a unary operator removal mutation. Insertions are formed by inserting the elements of the set {-, ¬, ++, ABS, -ABS,ZPUSH}, whenever appropriate.

Several operator mutants are intended to model the errors classified above. These operators produce mutants which are not strictly internal forms of any correct program, but are nonetheless useful in detecting certain categories of errors.

The first two operators are binary operators ⌈ and ⌉ which can stand instead of either arithmetic or logical operators. The effect of these operators is to evaluate both operands and to return either the right or left hand argument, ignoring the other one.

A second pair of binary operators, **trueop** and **falseop**, can be of boolean type only. These operators evaluate both operands and return either the constant value TRUE or FALSE, depending on which operator is applied.

There are several unary operators. Twiddle (denoted ++ or --) is an operator which returns its argument + 1 if the argument is an integer and $\pm$ .01% or .01 (whichever is greater) if the argument is real. The operator -ABS returns the negative of the absolute value. The ZPUSH(X) operators returns X if X is nonzero. However, if X is zero, ZPUSH by definition causes the mutant to be eliminated, thus forcing the expression X to be zero.

**Statement and Control Mutations:** A sequence of unlabelled non-decision statements in a program is called a <u>basic block</u>. It is a property of a basic blocks that if any one of the statements in a block is ever executed, then all statements in the block must also be executed.

One type of statement mutation determines whether or not the initial statement of each basic block is ever executed. The statement operators replaces the first statement of a basic block with a special statement called TRAP. The semantics of the TRAP statement is that if it is ever executed, it immediately causes the mutant to be eliminated. On the other hand, if such a mutant ever survives, then the corresponding basic block has never been executed. In this fashion, mutants can model a basic statement coverage measure of test data adequacy.

Statement coverage is strengthened by using a mutation operator which replaces each statement with a statement that has no effect, such as the Fortran CONTINUE statement. These mutants are designed to determine whether, in addition to being executed, the mutated statement has any effect on the program's execution.

A third statement operator changes the labels on control transfer statements and arithmetic conditionals to other labels which appear in the program.

The final statement operator to be discussed here modifies the structure of loops. One form of this operator changes the final label on Fortran DO loops to other labels which lie between the beginning of the loop and the end of the program. A second form of the operator changes the loop statement semantics. Recall, for example that the difference between a Fortran DO and an Algol FOR statement is that if the initial value of the FOR loop variable is smaller than the final value, the FOR loop is not executed, but a DO loop body is always executed at least once. Confusing this two loop constructs is a common programming error. A mutation operator that models such an error simply changes a DO statement to a FOR statement.

A set of mutant operators that is applicable to Fortran programs includes the following:

Operand Mutations

1. Constant Replacement (by +1, -1)
2. Scalar for Constant Replacement
3. Source Constant Replacement
4. Array Reference for Constant Replacement
5. Scalar Variable Replacement
6. Constant for Scalar Replacement
7. Array Reference for Scalar Replacement
8. Comparable Array Name Replacement
9. Constant for Array Reference Replacement
10. Scalar for Array Reference Replacement
11. Array Reference for Array Reference Replacement

Operator Mutations

12. Arithmetic Operator Replacement
13. Relational Operator Replacement
14. Logical Connective Replacement
15. Unary Operator Replacement
16. Unary Operator Removal
17. Unary Operator Insertion

Statement Mutations

18. Statement Execution (replacement by TRAP)
19. Statement Deletion
20. RETURN Statement Replacement

Control Structure Mutations

21. Jump Statement Replacement
22. DO statement Replacement

Adapting this set of operators to other languages involves analyzing the errors which can occur due to language features not present in Fortran. For example, to expand the Fortran operators to the simple Cobol subset discussed in Chapter 4, the following mutants should be considered.

Operand Mutations

1. Move implied decimal point in numeric items one place to the left or to the right.
2. Add or subtract one from an OCCURS clause count.
3. Insert FILLER of length one between two adjacent record items; also change FILLER lengths by one.
4. Reverse adjacent elementary items in records.
5. Alter file references.

Operator Mutations

6. Change ROUNDED TO truncation in arithmetic assignments
7. Change the sense of a MOVE

Control Structure Mutations

8. Interchange PERFORM and GOTO

We use the notation $\alpha ==> \beta$ to indicate the application of a mutant operator to construct $\alpha$ to produce mutation $\beta$. In general $\alpha$ can be a statement, group of statements, program or program fragment. If $\alpha$ is not a complete program, $\alpha ==> \beta$ is to be interpreted so that $\alpha$ is changed to $\beta$ and the remaining context of $\alpha$ remains intact if the result is a syntactically correct program.

**A Procedure for Developing Adequate Test Data**

Given a program P to test and a set of test data T, apply the mutant operator $\mu$ to obtain the set $\mu(P)$ of mutants. The first step is to execute the program P using test data. If P does not perform as specified on T, then certainly P is in error. If P performs as specified on T, we must determine whether T is adequate relative to $\mu(P)$. Only two possibilities arise.

1. a mutant $Q \varepsilon \mu(P)$ gives different results from P, or
2. a mutant $Q \varepsilon \mu(P)$ gives the same results as P.

In case (1), Q is said to be <u>dead</u>, while in case (2), the mutant is called <u>live</u>. Obviously, if T leaves only live mutants that are equivalent to P, m(P,T)=1, and therefore T is adequate relative to the set of mutants. If T leaves live, nonequivalent mutants, then either T can be augmented by some test strategy to an adequate (relative to $\mu(P)$) test set, or there is an error in P that has not yet been revealed.

It is not apparent from this description that the procedure is either feasible or effective in detecting errors. As we will show in later chapters, there is a methodology for implementing this procedure which makes it computationally attractive. By the same token, we will demonstrate the error detection capabilities of this procedure. In lieu of these developments, however, the reader should notice that we have outlined a principle which can provide inferences of correctness. The inductive strength of those inferences is directly related to a single set of experimental observations — the observations which support the Competetent

Programmer Assumption with a specified degree of confidence.

## Error Coupling

A coupling effect asserts that test data that is sensitive enough to cause all simple mutants to fail is also sensitive enough to cause all complex mutants to fail. Note that error coupling is not a provable phenomenon in a mathematical sense; indeed, there are very simple counterexamples to it. It is, however, a useful principle that can be observed to hold for broad classes of programs and which can be measured in typical programming environments.

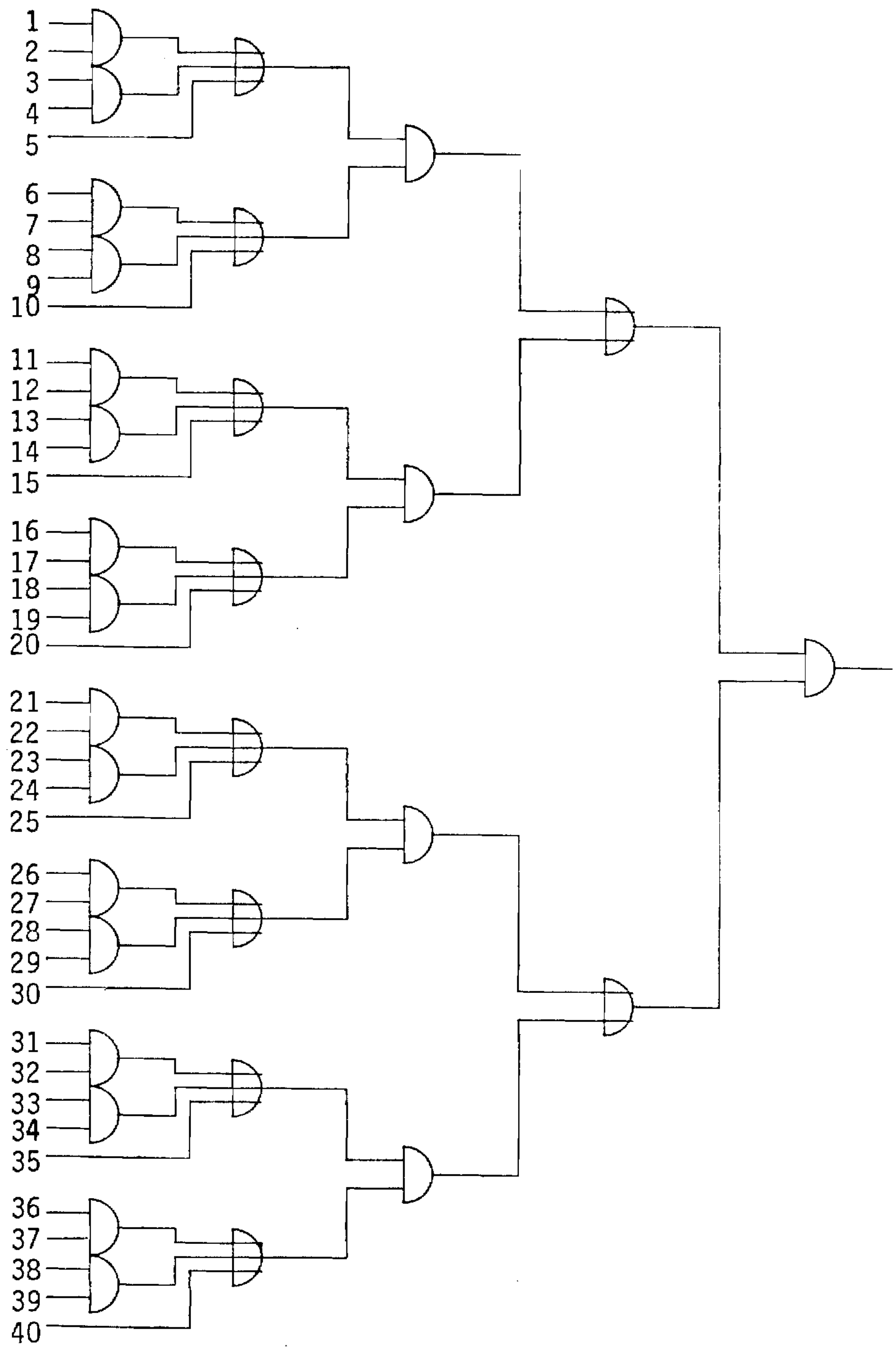
Since error classifications result in sets of mutants, it may help to define error coupling in terms of mutant operators.

**Definition:** Let $\mu(P)$ and $\phi(P)$ define sets of mutants for each P in a programming system. Then $\mu$ is said to be coupled to $\phi$ if $m_\mu(P,T) = 1$ implies $m_\phi(P,T) = 1$.

It may have occurred to the reader that program mutation is the software version of fault detection: that is the origin of a hypothesized coupling effect. The fault detection problem may be specified as follows. Given a digital circuit C and Boolean function f (the specification of the circuit), determine whether or not the circuit C realizes the function f. A natural way of solving a fault detection problem is to submit inputs to C. If C works as expected then the circuit is most likely to be fault-free. Suppose C determines the complement of a 32 bit number. Exhaustive testing of an arbitrary circuit might require as many as $2^{32}$ inputs.

However, the faults (or errors) that are assumed to occur are usually constrained in some way. For example, it is commonly assumed that all faults are of the form: a single wire is permanently "stuck at" 0 or 1. These are called <u>single faults</u>. The single fault assumption reduces the number of test case to under 100. Such assumptions are derived on the basis of experience, the independence of the components of C and the statistical analysis of similar circuits. Using a single fault assumption in a given fault detection problem, a tester obtains a test set I such that C performs correctly on I and no other single fault circuit performs correctly on I. Then either C is correct or it is not in the set of single fault circuits for a circuit correctly realizing f.

The problem that arises in fault detection is how close a single fault test set comes to detecting multiple faults which might actually occur (circuit testers call this phenomenon <u>co</u>verage of the multiple faults). In many circumstances single fault tests sets provably cover many or all multiple faults. For example, there are classes of circuits (e.g., cascaded two-level networks and internal fanout-free networks) such that if I is a set of test data which solves the single fault detection problem on a given set of k wires, then I also solves all multiple fault detection problems on those wires. As a concrete example, consider the combinational logic circuit shown in Figure 2 below.


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
33
34
35
36
37
38
39
40

Let $K = \{1,3,6,8,11,13,16,18,21,23,26,28,31,33,36,38\}$ denote the indicated 16 inputs of the circuit, and let I be the test set of 56 input vectors shown in Table 4. The entries under i denote the number of the input vector. The vector and parity entries must be read together to determine the value of the vector. For example an entry with vector entry $\alpha_1,\alpha_2,\alpha_3$ and parity entry $\beta \in \{0,1\}$ denotes an input vector in which inputs numbered $\alpha_i$, $1 \leq i \leq 3$, are set to $\beta$ and the remaining inputs are set to $\beta$+1 mod 2.

| i | Parity | Vector | i | Parity | Vector |
|---|---|---|---|---|---|
| 1 | 0 | 3,5,11,13,15 | 29 | 1 | 12,18,19,31,32,38,39 |
| 2 | 0 | 1,5,11,13,15 | 30 | 1 | 14,16,17,33,34,36,37 |
| 3 | 0 | 8,10,16,18,20 | 31 | 1 | 13,14,17,33,34,36,37 |
| 4 | 0 | 6,10,16,18,20 | 32 | 1 | 11,12,19,31,32,38,39 |
| 5 | 0 | 1,3,5,13,15 | 33 | 1 | 1,8,9,21,22,28,29 |
| 6 | 0 | 1,3,5,11,15 | 34 | 1 | 3,6,7,23,24,26,27 |
| 7 | 0 | 6,8,10,18,20 | 35 | 1 | 3,4,6,23,24,26,27 |
| 8 | 0 | 6,8,10,16,20 | 36 | 1 | 1,2,8,21,22,28,29 |
| 9 | 0 | 1,2,3,4,11,13,15 | 37 | 1 | 11,18,19,31,32,38,39 |
| 10 | 0 | 6,7,8,9,16,18,20 | 38 | 1 | 13,16,17,33,34,36,37 |
| 11 | 0 | 1,3,5,11,12,13,14 | 39 | 1 | 13,14,16,33,34,36,37 |
| 12 | 0 | 6,8,10,16,17,18,19 | 40 | 1 | 11,12,18,31,32,38,39 |
| 13 | 0 | 23,25,31,33,35 | 41 | 1 | 1,2,8,9,22,28,29 |
| 14 | 0 | 21,25,31,33,35 | 42 | 1 | 3,4,6,7,24,26,27 |
| 15 | 0 | 28,30,36,38,40 | 43 | 1 | 3,4,6,7,23,24,27 |
| 16 | 0 | 26,30,36,38,40 | 44 | 1 | 1,2,8,9,21,22,29 |
| 17 | 0 | 21,23,25,33,35 | 45 | 1 | 11,12,18,19,32,38,39 |
| 10 | 0 | 21,23,25,31,35 | 46 | 1 | 13,14,16,17,34,36,37 |
| 19 | 0 | 26,28,30,38,40 | 47 | 1 | 13,14,16,17,33,34,37 |
| 20 | 0 | 26,28,30,36,40 | 48 | 1 | 11,12,18,19,31,32,39 |
| 21 | 0 | 21,22,23,24,31,33,35 | 49 | 1 | 1,2,8,9,21,28,29 |
| 22 | 0 | 26,27,28,29,36,38,40 | 50 | 1 | 3,4,6,7,23,26,27 |
| 23 | 0 | 21,23,25,31,32,33,34 | 51 | 1 | 3,4,6,7,23,24,26 |
| 24 | 0 | 26,28,30,36,37,38,39 | 52 | 1 | 1,2,8,9,21,22,28 |
| 25 | 1 | 2,8,9,21,22,28,29 | 53 | 1 | 11,12,18,19,31,38,39 |
| 26 | 1 | 4,6,7,23,24,26,27 | 54 | 1 | 13,14,16,17,33,36,37 |
| 27 | 1 | 3,4,7,23,24,26,27 | 55 | 1 | 13,14,16,17,33,34,36 |
| 28 | 1 | 1,2,9,21,22,28,29 | 56 | 1 | 11,12,18,19,31,32,38 |

Table 4. Single Fault Test I

It can be shown that I also covers every multiple fault involving every k-tuple of the lines from K, for k=2,3. Furthermore, I covers 90% of the multiple faults involving m of these lines for m=4,5,6. For multiple faults simultaneously involving all 16 wires, however, less than half of the $2^{16}$ faults are covered. It is essentially a problem in electrical engineering to determine whether or not k simultaneous faults are likely for $k \leq 6$. If so, then it would seem appropriate to use the 56 test vectors in I.

The coupling of errors in programs has much in common with the notion of test set coverage. It appears that test data which is adequate for simple errors is also adequate for many complex errors. In fact, the assumptions made about the programming process in Chapter 1 give us some hope that error coupling in programs is a stronger effect than coverage of multiple faults in digital circuits. A fault in a circuit is an event of nature -- it is essentially random. However, since programs are not created randomly, it seems unlikely that errors are created randomly Neither are errors created by an adversary. Rather, errors are introduced, corrected and reintroduced by programmers diligently creating programs which they intend to be error-free. The result of this activity is that errors are not created specifically to <u>avoid</u> error coupling. There is a great deal of information sharing within a program, and textually distant source statements can exert subtle influences on each other during program execution. The net effect of this interdependence is that complex errors can make their presence known through their effects on single statements and single syntactic items within those statements. Hence, a test that deals with an an error through a simple mutant in one portion of a program can implicitly reveal errors in portions of the program that depend or affect the statement to which the mutant is explicitly applied. Test set coverage also illustrates a theme that runs through our treatment of the coupling effect: the interplay between subcases for which simple errors cover complex errors and statistical estimates for the general case.

We will illustrate this principle with a simple example. Consider the Fortran program B7 for computing statistics from a table of observations.

```
      SUBROUTINE TAB1(A,NV,NO,NINT,S,UBO,FREQ,PCT,STATS)
      INTEGER INTX
      REAL TEMP,SCNT,SINT
      INTEGER INN,J,IJ
      REAL VMAX,VMIN
      INTEGER I,NOVAR
      REAL WBO(3),STATS(5),PCT(NINT),FREQ(NINT)
      REAL UBO(3),S(NO)
      INTEGER NINT,NO,NV
      REAL A(600)
      NOVAR = 5
      DO 5 I=1,3
 5    WBO(I)=UBO(I)
      VMIN = 0.1000000000E+11
      VMAX =- 0.1000000000E+11
      IJ=NO*(NOVAR-1)
      DO 30 J=1,NO
      IJ = IJ+1
      IF(S(J)) 10,30,10
 10   IF(A(IJ)-VMIN)15,20,20
 15   VMIN = A(IJ)
 20   IF(A(IJ)-VMAX)30,30,25
 25   VMAX = A(IJ)
 30   CONTINUE
      STATS(4) = VMIN
      STATS(5) = VMAX
      IF(UBO(1)-UBO(3)40,35,40
 35   UBO(1) = VMIN
      UBO(3) = VMAX
 40   INN = UBO(3)
      DO 45 I=1,INN
      FREQ = 0.0000
 45   PCT(I) = 0.0000
      DO 50 I=1,3
 50   STATS(I) = 0.0000
      SINT = ABS((UBO(3)-UBO(1))/(UBO(2)-2.0000))
      SCNT = 0.0000
      IJ = NO*(NOVAR-1)
      DO 75 J=1,NO
      IJ = IJ+1
      IF(S(J))55,75,55
 55   SCNT = SCNT+1.0000
      STATS(1) = STATS(1)+A(IJ)
      STATS(3) = STATS(3)+A(IJ)*A(IJ)
      TEMP = UBO(1)-SINT
      INTXT = INN-1
      DO 60 I=1,INTXT
      TEMP = TEMP+SINT
      IF(A(IJ)-TEMP)70,60,60
```

```
60    CONTINUE
      IF(A(IJ)-TEMP)75,65,65
65    FREQ(INN) = FREQ(INN)+1.0000
      GO TO 75
70    FREQ(I) = FREQ(I)+1.0000
75    CONTINUE
      IF(SCNT)79,105,79
79    DO 80 I=1,INN
80    PCT(I) = (FREQ(I)*100.0000)/SCNT
      IF(SCNT-1.0000)85,85,90
85    STATS(2) = STATS(1)
      STATS(3) = 0.0000
      GO TO 95
90    STATS(2) = STATS(1)/SCNT
      STATS(3) = SQRT(ABS((STATS(3)-(STATS(1)*STATS(1)/
    * SCNT)/(SCNT-1.0000)))
95    DO 100 I=1,3
100   UBO(I) = WBO(I)
105   RETURN
      END
```

This program is adapted from a collection of statistical and scientific programs and contains an artificially inserted error. An error occurs in the line that reads

```
40 INN = UBO(3).
```

The statement should be

```
40 INN = UBO(2).
```

Consider,

the mutant

```
IF (A(IJ) - TEMP)75,65,65  ==>  IF (A(IJ) - 1.000)75,65,65
```

Control reaches this point only if A(IJ) is bigger than TEMP, so control always passes to 65. By tracing the flow of control we discover that TEMP is equal to the value of the input parameter UBO(3) at this point. To eliminate this mutant, then, we must find a value where A(IJ) is less than one but larger than UBO(3). Therefore UBO(3) must be less than one. There is nothing in the specifications that rules out UBO(3)'s being less than one, but the error causes UBO(3) to be assigned to the integer variable INN. All the feasible paths that go through the mutated statement also go through label 65, which references FREQ(INN). Since INN is less than or equal to zero, an array index out of bounds error is detected.

As we have already mentioned, there is no useful sense in which errors are <u>provably</u> coupled in real programs. Therefore, it makes sense to inquire into the <u>extent</u> to which errors are coupled.

**Definition:** Let P be a program and consider $\mu(P)$ and $\phi(P)$ as defined above. We will say that $\mu$ is coupled to $\phi$ with coupling coefficient $(1-\omega)$ if $\omega$ is the largest number such that for any test set T with m[sub $\mu(P,T) = 1$ $\phi(P)-\Delta_{\phi}(P,T) \leq \omega|\phi(P)|$.

We plan on using this definition in experimental investigations into the coupling effect. The goal of these investigations is to determine whether or not a tester can assume with a reasonable degree of confidence that test data which is adequate for simple mutants is also adequate for mutants which explicitly satisfy the competent programmer assumption. Examining all possible test cases is not feasible, so this definition needs some modification to be experimentally useful. We will ,therefore, usually work with another coefficient, z.

**Definition:** The coefficient z is the fraction of the nonequivalent members of $\phi$ that are not killed by some particular test case.

z is then a random variable distributed over the space of pairs (P,T), where P is a program, and T is adequate relative to $\mu(P)$. Clearly $\omega$ is an upper bound on z. An experiment on the coupling effect is a measurement of the strength of that effect by measurement of z. The measurement of z is in turn, an estimate on $\omega$. In practice, z itself can only be estimated by sampling. The usual case is that we will determine a confidence interval for z. The conclusion of an experiment organized in this way will then be of the following form. For programs selected from a given population and test data generated by process G (adequate for $\mu$) the values of z were estimated by sampling from $\phi$ and found to range between x and y.

Thus, if the population from which we sample is similar to the population of programs about which we want to make quantitative estimates, and G is the method available for generating test data whose strength we want to determine, and if $\phi$ is an estimate of the distribution of likely mutants, we can use the estimated values of z to bound the probability that errors remain in a given program.

## Bibliographic Notes

The Competent Programmer Assumption was first articulated by DeMillo, Lipton and Sayward [DeMillo, 1978a]. The concept was refined and related to the correctness of mutation testing in a series of papers which followed [Acree, 1979], [Acree, 1980], [Budd, 1980].

The treatment of error data and data gathering over the past decade has been surveyed by Gannon [Gannon, 1983]. See also Thibodeau [Thibodeau, 1982] for a critical evaluation of existing data gathering efforts. The data cited in this chapter was taken from [Youngs, 1974] and [Thayer, 1978].

The form of many of the mutant operators presented above was implicit in [Budd, 1978b]. As experience with constructing automated systems grew, many new operators which are sensitive to specialized error conditions or language features were designed. The background on these designs can be found in [Acree, 1979], [Acree, 1980], [Budd, 1980], and [Hanks, 1980].

The notion of error coupling was proposed in [DeMillo, 1978a]. Budd's thesis [Budd, 1980] and several subsequent papers have (see, e.g., [DeMillo, 1979]) have given heuristic arguments which support error coupling in software. The operational definitions of coupling coefficients are due to Acree [Acree, 1980]. Experimental justifications for coupling are discussed in Chapter 6. The example used for logic circuit test set coverage appeared in a paper by Agarwal and Masson [Agarwal, 1979] in which an number of special cases of single fault coverage of multiple faults are derived along with a general technique for calculating test coverage.

# Chapter 3

## Theoretical Studies

There are two possible approaches to applying mutation: (1) For fixed programming system P define the mutants of P in terms of syntactic and semantic transformation rules that alter P's syntax and interpretation in a way that formally reflects the errors a competent programmer could have made in producing P, or (2) define $\mu$ = P. Notice that, by virture of Theorems 1.20 and 1.21, (2) has the effect of reducing test data adequacy relative to a set of errors to simple test data adequacy. For theoretical studies, (2) is often the more tractable approach since many useful properties of programs can be inherited from their programming systems.

We recall the following fact from Chapter 1:

**Theorem 1.18:** The following statements are equivalent. (a) the $\mu$(P)-adequate decision problem is solvable. (b) there is a computable $\mu$(P) test strategy. (c) the $\mu$-equivalence problem is decidable.

Then the following corollary is immediate.

**Corollary:** If there is a computable test strategy to generate $\mu$(P) adequate test data T, then the equivalence of P and any program Q in $\mu$(P) must be decidable.

At first glance the result of this theorem appears to cast serious doubt on our ability to derive any interesting positive results, since the equivalence problem is undecidable for most interesting language classes. As will be seen in this chapter, however, we can obtain useful theoretical results by choosing the set $\mu(P)$ to capture some special properties of the original program P.

For the remainder of this chapter we will consider two specific programming systems: decision tables and LISP programs.

## Decision Tables.

A <u>decision table</u> is a structured way of describing decision alternatives. Decision tables are mainly used for data processing applications although from time to time they have been suggested as tools for certain analytic studies and for organizing test data selection predicates.

A decision table is composed of a set of conditions, a set of actions, and a table divided into two parts. Entries in the upper part are chosen from the set {YES, NO, DON'T CARE} (denoted Y, N, and * ); entries in the lower table are either DO or DON'T DO (denoted X and 0). Each column in the matrix is called a rule. An example is shown in Figure 1.

| | RULES 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| condition 1 | Y | Y | N | * |
| condition 2 | N | * | Y | Y |
| condition 3 | * | Y | Y | N |
| condition 4 | N | Y | * | * |
| action 1 | X | X | 0 | X |
| action 2 | X | 0 | 0 | 0 |
| action 3 | 0 | 0 | X | X |

Figure 1.
A Typical Decision Table

To execute such a program on an input, the conditions are first simultaneously evaluated, forming a vector of YES-NO entries. This vector is then compared to every rule. If the vector matches any rule, the indicated actions are performed. If, for each possible data item, there is at least one rule that can be satisfied, we say the decision table is <u>complete</u>. We say it is <u>consistent</u> if there is at most one rule.

**Definition:** Let P be a decision table with rules $R_1,\ldots,R_n$, and for each $x \in D$, the domain of P, let $v(x)$ be a sequence with values in the set {YES,NO} such that $v(x)_i$ is the value of condition i when evaluated on input x. Rule $R_j$ $(1 \leq j \leq n)$ is said to be satisfied by input x if whenever $R_{ji} \in \{YES,NO\}$, $R_{ji} = v(x)_i$.

**Definition:** Let **P** be a decision table with domain **D**. **P** is a complete decision table if for all $x \in$ **D**, there is at least one rule of P that is satisfied by x.

**Definition:** Let P be a decision table with domain **D**. P is a consistent decision table if for all $x \in$ **D**, there is at most one rule of P that is satisfied by x.

We define the programming system **P** to be the set of consistent decision tables. In this case, the behavior of programs on **D** can be characterized functionally. Without loss of generality, we assume that **P** consists of complete decision tables, since an incomplete decision table can always be simulated by a complete decision table by adding actions that return error flags and rules that are satisfied by previously unmatched inputs in such a manner that the domain of the incomplete table is consistently extended to all of **D**.

Without loss of generality, we may also assume that no two rules specify exactly the same set of actions. Suppose that P is a decision table with two such rules R and R'. Then by the addition of at most one new condition to **P**, R and R' can be combined into a single rule. With this assumption, we can -- given an example of input-output behavior — always determine which rule was applied to give the required output.

**Definition:** For each P ε **P**, we define a set of mutants of P as follows: $\phi(P) \subseteq \mathbf{P}$ is the set of all consisent decision tables having the same conditions and actions as P.

Notice that the mutants of P differ from P only in the tabular portion of the program. The number of rules may be different, the assignment of actions to satisfied rules need not be correlated, and the occurrences of YES, NO and * entries may be unrelated. This notion of mutant program models the concept of an aribtrary coding error in a decision table: since the conditions and actions must be preserved, it is assumed that the source of errors is not in understanding requirements or specifications, but rather in implementing the sequences of actions to be invoked.

**Definition:** For each P ε **P**, the set of <u>simple</u> mutants of P, $\mu(P) \subseteq \phi(P)$ is defined as follows: $P' \in \mu(P)$ if P′ is a mutant of P such that if some entry $R_{ij}$ in rule i of P is *, then the corresponding entry $R'_{ij}$ in rule i of P′ is either YES or NO and all other rules and actions are identical.

The simple mutants of P are those members of $\phi$ that are formed by changing a <u>single</u> * entry into either a YES or NO entry. If P is consistent then all simple mutants are consistent. Some of these mutants may be equivalent to P. The mutant that changes position j in rule i from a * to a Y is equivalent to P only if it is impossible for any input to satisfy rule i and not satisfy this condition.

Suppose we test decision table programs by applying Theorem 1.21. That is, we determine the relative adequacy of a test set by computing the mutation score of the test set for a given set of mutants. By naively modelling all possible errors, we have a mutant set $\phi(P)$ that can be as large as $3^n + 2^m$, if P has n conditions and m actions. Since each mutant in $\phi(P)$ could require a distinct test set to distinguish it from P, the number of tests required in a test set adequate relative to $\phi(P)$ could be exponential in the size of P. On the other hand, there are at most two simple mutants for every table entry in P. This means there are no more than 2nm simple mutants. Each mutant requires at most a single test case to differentiate it from P. Therefore, even though there are potentially $2^n$ different inputs, a test set that is adequate relative to $\mu(P)$ need have only at most 2nm inputs.

Since $\phi$ models arbitrary coding errors while $\mu$ models a rather more restricted class of errors, the relative advantage computing the mutation score on the set of simple mutants cannot really be exploited unless there is a coupling of simple and complex errors for programs in **P**.

Our goal will be to derive a provable coupling effect for the programming system **P**. In particular, we wish to show that if $m_\phi$ and $m_\mu$ are the mutation scores computed over $\phi(P)$ and $\mu(P)$, respectively, then for all $P \in$ **P**,

$$m_\phi(P,T) = 1 \text{ if and only if } m_\mu(P,T) = 1.$$

Assume we have such a set T. We require that T satisfy a minimal test requirement, the decision table analog of statement coverage. We will assume that every rule in P is satisfied at least once by some member of T, adding points if necessary to meet this condition. If all rules contain *'s, then this condition is met initially.

This condition on T can be insured in test sets adequate relative to a rich enough mutant set. Indeed, if $\phi$ had been defined to allow modifications to the actions of decision tables, then it would have been possible to define $\phi$ so that $m_\phi(P,T) = 1$ only if T satisfies each rule of P at least once. This expansion of $\phi$ does not change the error coupling properties of $\mu$, but it would add considerable complexity to the arguments to follow.

**Definition:** Let P and Q be decision tables, $Q \in \phi(P)$, and let T be a test set. If $P^*(T) = Q^*(T)$, then Q is said to <u>test equal</u> to P on T.

Since each rule in P has a unique set of actions, it follows by a simple counting argument that, if Q tests equal to P, then for each rule in P there is a corresponding rule in Q with exactly the same actions. Using this fact, we can show the following:

**Theorem 1:** Suppose $m_\mu(P,T)=1$, and Q tests equal to P (on T). Let $V(P)_i$ be the set of inputs satisfying rule $R_i$ if P and let $V(Q)_i$ be the set of input satisfying the corresponding rule of Q. Then $V(P)_i \subset V(Q)_i$.

**Proof:** First note that it is not possible for a rule to have a Y entry in P and for the corresponding rule in Q to have an N, or vice versa. Otherwise, no data that satisfied the rule in P could satisfy the rule in Q.

Consider each * entry in P. There are two cases. If the change that replaces this * by a Y (the same argument holds for N) results in an equivalent program, then the conjunction of the other conditions implies a YES in this position. In this case, it doesn't matter whether Q has a Y or a * (and these are the only two possibilities) — this change cannot contribute to decreasing the size of the set $V(Q)_i$. On the other hand, if this change does not result in an equivalent mutant, then D contains points that satisfy the rule and both satisfy and fail to satisfy this particular condition. Both these must be accepted by the same rule in Q. Therefore Q must also have a * in this position.

The only remaining possibility is that some rule $R_i$ in P has a Y (or N) and the corresponding position in Q has a *. This strictly increases the size $V(Q)_i$, giving our result. []

**Theorem 2:** Let $P \varepsilon$ P and let T be a test set. If $m_\mu(P,T)=1$, then $m_\phi(P,T) = 1$.

**Proof:** Let $V(P)_i$ be the set of inputs satisfying rule $R_i$ in P. Since P is consistent, the $V(P)_i$ are disjoint. Since P is complete, they cover the entire space of inputs. Each rule in Q must be satisfied by at least the set satisfying the corresponding rule in P. Since Q is consistent, it can satisfy no more. []

Recall that Theorem 1.18 stated that we could form an adequate test set relative to the set of mutants only if we could decide equivalence of P and each of its mutants. Obviously there are some cases where this is true (for example, when all conditions are independent and therefore none of the mutants are equivalent). We can easily find examples where this is not true. Consider, for example, two conditions where the implication

$$\text{condition}_1 \Rightarrow \text{condition}_2$$

is undecidable, and construct a decision table as shown in Figure 2.

| | |
|---|---|
| *condition 1* | *Y* |
| *condition 2* | * |
| *print "YES"* | *X* |

Figure 2.
Example of Undecidable Equivalence

We can replace the * in the condition 2 row with a Y if and only if condition 1 always implies condition 2. In this fashion using almost any undecidable question we can construct a program with the property that the equivalence question for it and one of its mutants is undecidable.

The most restrictive assumption made in proving Theorem 2 seems to be that each rule must have a distinct set of actions. To show that this restriction cannot be eliminated altogether, consider the two decision tables shown in Figure 3. The two programs are not equivalent (they process the input NNYN differently), yet they agree on a set of test inputs {NNYY,NYYN,YYNN,YNNY,NNNN,NYNY,YYYY,YNYN}, which is adequate relative to $\mu(P)$.

Program P

| | | | |
|---|---|---|---|
| N | Y | N | Y |
| * | * | * | * |
| Y | N | N | Y |
| * | * | * | * |
| X | X | 0 | 0 |
| 0 | 0 | X | X |

Program Q

| | | | |
|---|---|---|---|
| * | * | * | * |
| N | Y | N | Y |
| * | * | * | * |
| Y | N | N | Y |
| X | X | 0 | 0 |
| 0 | 0 | X | X |

Figure 3.
A Case not Covered by Mutation

It is not known whether the restriction to rules having distinct actions can be replaced with a weaker assumption, or whether there is any test method that can be used to demonstrate correctness in this case other than trying all $O(2^n)$ possibilities.

## Lisp Programs

In this section we will consider the programming system P consisting of programs written in the subset of LISP containing the functions CAR, CDR, and CONS and the predicate ATOM.

We will refer to S-expressions as <u>points</u>. We assume that all points have unique atoms. Clearly if two programs agree on all points then they are equivalent over the entire domain, so there is no generality lost in this assumption.

**Definition:** A LISP program is a <u>selector</u> program if it is composed of just CAR and CDR. We inductively define a <u>straight-line</u> program as a selector program or a program formed by the CONS of two other straight-line programs.

**Straight-line programs:** We will show in this section that in the subsystem consisting of straightline programs, if $\mu$ is the constant mapping onto the entire subsystem, then $m_\mu(P,\{X\}) = 1$, provided only that X is a point such that P(X) is defined.

We first note that the power of a selector program is very weak.

**Theorem 3:** If two selector programs test equal on any input for which they are both defined, they must compute identical values on all points.

**Proof:** The only power of a selector program is to choose a subtree out of its input and return it. We can view this process as selecting a position in the complete CAR/CDR tree and returning the subtree rooted at that position. Since there is a unique path from the root to this position, there is a unique predicate that selects it. Since atoms are unique, by merely observing the output we can determine the subtree that was selected. []

**Definition:** A straight-line program P(X) is <u>well formed</u> if for every occurrence of the construction CONS(A,B) it is the case that A and B do not share an immediate parent in X.

The intuitive idea of this definition is that a program is well formed if it does not do any more work than it needs to. Notice that being well formed is a structural property of programs.

We now define a complexity measure for straight-line programs.

**Definition:** The <u>CONS-depth</u> of a program is defined inductively.

1. The CONS-depth of a selector program is zero.
2. The CONS-depth of a straight-line program

   P(X) = CONS(P1(X),P2(X))

   is

   1 + MAX(CONS-depth(P1(X)),CONS-depth(P2))).

**Theorem 4:** If two well formed selector programs test equal on any point for which they are both defined, then they must have the same CONS-depth.

**Proof:** Assume we have two programs P and Q and a point X such that P(X) = Q(X), yet the CONS-depth(P) < CONS-depth(Q). This implies that there is at least one subtree in the structure of Q that was produced by CONSing two straight-line programs while the same subtree in P(X) was produced by a selector. But then the objects Q CONSed must have an immediate ancestor in X, contradicting the fact the Q is well formed. []

**Theorem 5:** If two well formed straight-line programs test equal on any point X for which they are both defined, then they must test equal on all points.

**Proof:** The proof will be by induction on the CONS-depth. By Theorem 4, any two programs that agree on X must have the same CONS-depth. By Theorem 3 the theorem is true for programs of CONS-depth zero. Hence, we will assume it is true for programs of CONS-depth n and show the case for n+1.

If program P has CONS-depth n+1 then it must be of the form CONS(P,Q) where P and Q have CONS-depth no greater than n. Assume we have two programs P and Q in this fashion. Then for all Y:

P(Y) = Q(Y) if and only if

CONS(P1(Y),P2(Y)) = CONS(Q1(Y),Q2(Y)) if and only if

P1(Y) = Q1(Y) and P2(Y) = Q2(Y)

Hence by the induction hypothesis P and Q must test equal for all Y. []

We can easily generalize Theorem 5 to the case where we have multiple inputs. Recall that each atom is unique; therefore given a vector of arguments we can form them into a list and the result will be a single point with unique atoms. Similarly, a program with multiple arguments can be replaced by a program with a single argument by assuming the inputs are delivered in the form of a list, and replacing each occurrence of an argument name with a selector function accessing the appropriate position in this list. Using this construction and assuming that Theorem 5 does not hold in the case of multiple arguments, it is possible to construct two programs with single arguments for which Theorem 5 fails, giving a contradiction.

To summarize this section: for any well formed straight-line program, any unique atomic point for which the function is defined is adequate to differentiate the program from all other well formed straight line programs.

**Recursive programs:** The type of programs we will study in this section can be described as follows. The input to the program will consist of <u>selector variables</u>, denoted $x_1,\ldots,x_m$, and <u>constructor variables</u>, denoted $y_1,\ldots,y_p$. A program will consist of a program body and a recurser. A <u>program body</u> consists of n statements, each statement composed of a <u>predicate</u> of the form ATOM$(t(x_1))$ where t is a selector function and $x_1$ a selector variable, and a straight-line <u>output function</u> over the selector and constructor variables. A <u>recurser</u> is divided into two parts. The <u>constructor</u> part is composed of p assignment statements for each of the p constructor variables where $y_i$ is assigned a sraight-line function over the selector variables and $y_i$. The selector part is composed of m assignment statements for the m selector variables where $x_i$ is assigned a selector function of itself.

The example in Figure 4 should give a more intuitive picture of this class of programs. Given such a program, execution proceeds as follows: Each predicate of the execution; otherwise if any predicate is TRUE the result of execution is the associated output function. Otherwise, if no predicate evaluates TRUE then the assignment statements in the recurser and constructor are performed and execution continues with these new values.

$$\text{Program } P(x_1, \ldots, x_m, y_1, \ldots, y_p) =$$

$$
\begin{array}{l}
IF\ p_1(x_{i1})\ THEN\ f_1(x_1, \ldots, x_m, y_1, \ldots, y_p) \\
\quad ELSE\ IF\ \ldots \\
\quad \ldots \\
\quad ELSE\ IF\ p_n(x_{in})\ THEN\ f_n(x_1, \ldots, x_m, y_1, \ldots, y_p) \\
\qquad ELSE \\
\qquad y_1 := g_1(y_1, x_1, \ldots, x_m) \\
\qquad y_p := g_p(y_p, x_1, \ldots, x_m) \\
\qquad x_1 := n_1(x_1) \\
\qquad x_m := n_m(x_m) \\
\qquad P(x_1, \ldots, x_m, y_1, \ldots, y_p)
\end{array}
$$

Figure 4.
A Recursive Program

We will make the following restrictions on the programs we will consider:

1. All the recursion selector and recursion constructor functions must be non-trivial.
2. Every selector variable must be tested by at least one predicate.
3. There is at least one output function that is not a constant.
4. (Freedom) For each $1 < k \leq n$ and $\lambda \geq 0$ there exists at least one input that causes the program to recurse $\lambda$ times before exiting with output function k.

Let $\phi$ be the set of all programs with the same number of selector and constructor variables as P, the same number of predicates, and output functions no deeper than some fixed limit olimit. Our goal is to construct a set of test cases T that is adequate relative to $\phi$. The set of simple mutants $\mu$ will be described in the course of the proof, as they enter into the arguments. The proof will proceed in several smaller steps: We first give some basic definitions and demonstrate some tools that we will use in later sections. We then show how to use testing to bound the depth of the selector functions. We then narrow the form of the selector functions still further, and finally show that they must exactly match P. In preparation for the main theorem, we first deal with the points tested by the predicates.

As in the previous section, we will use capital letters from the end of the alphabet to represent vectors of inputs. Hence we will refer to P(X) rather than $P(x_1,...,x_m,y_1,...,y_p)$. Similarly we will abbreviate the simultaneous application of constructor functions by C(X) and recursion selectors by R(X).

We will use letters from the start of the alphabet to represent <u>positions</u> in a variable, where a position is defined by a finite CAR-CDR path from the root. When no confusion can arise we will frequently refer to "position a in X", whereby we mean position a in some $x_i$ or $y_i$ in X. We will sometimes refer to position b relative to position a, by which we mean to follow the path to a and starting from that point follow the path to b.

The <u>depth</u> of a position will be the number of CARs or CDRs necessary to reach the position starting from the root. Similarly the depth of a straight-line function will be the deepest position it references, relative to its inputs. Let w be the maximum depth of any of the selector, constructor, recurser, or output functions in P. The <u>size</u> of an input X will be the maximum depth of any of the atoms in X.

We can extend the definition of $\lesssim$ to the space of inputs by saying X $\lesssim$ Y if and only if all the selector variables in X are smaller than their respective variables in Y, and similarly the constructor variables. We will say Y is X "pruned" at position a if Y is the largest input less than or equal to X in which a is atomic. This process can be viewed as simply taking the subtree in X rooted at a and replacing it by a unique atom.

If a position (relative to the original input) is tested by some predicate we will say that the position in question has been <u>touched</u>. Call the n positions touched by the predicates of P without going into recursion the <u>primary positions</u> of P.

The assumption of freedom asserts only the existence of inputs X that will cause the program to recurse a specific number of times and exit by a specific output function. Our first theorem shows that this can be made constructive.

**Theorem 6:** Given $\lambda \geq 0$ and $1 \lesssim i \lesssim n$ we can construct an input X so that P(X) is defined and when given X as an input P recurses $\lambda$ times before exiting by output function 1.

<u>Proof</u>: Consider m+p infinite trees corresponding to the m+p input variables. Mark in BLUE every position that is touched by a predicate function and found to be non-atomic in order for P to recurse $\lambda$ times and reach the predicate i. Then mark in RED the point touched by predicate i after recursing $\lambda$ times.

The assumption of freedom implies that no BLUE vertex can appear in the infinite subtree rooted at the RED vertex, and that the RED vertex cannot also be marked BLUE. Now mark in YELLOW all points that are used by constructor functions in recursing $\lambda$ times, and each position used by output function i after recursing $\lambda$ times. The assumption of freedom again tells us that no YELLOW vertex can appear in the infinite subtree rooted at the RED vertex. The RED vertex may, however, also be colored YELLOW, as may the BLUE vertices.

It is a simple matter then to construct an input X so that

1. all BLUE vertices are interior to X (non-atomic),
2. the RED vertex is atomic, and
3. all YELLOW vertices are contained in X (they may be atomic). []

Notice that the procedure given in the proof of Theorem 6 allows one to find the <u>smallest</u> X such that the indicated conditions hold. If a is the implies that no point can be twice touched; hence the minimal a point is a well defined concept.

Given an input X such that P(X) is defined, let $F_X(Z)$ be the straight-line function such that $F_X(X) = P(X)$. Note that by Theorem 5, $F_X$ is defined by this single point.

**Theorem 7:** For any X for which P(X) is defined, we can construct an input Y with the properties that P(Y) is defined, $Y \geq X$ and $F_X \neq F_Y$.

**Proof:** Let $\lambda$ and i be the constants such that on input X, P recurses $\lambda$ times before exiting by output function i. Let the predicate $p_i$ test variable $x_j$.

There are two cases. First assume f is not a constant function. Now it is possible that the position that would be tested by $p_i$ after recursing $\lambda+1$ times is an interior position in X, but since X is bounded there must be a smallest $k > \lambda$ such that the predicate $p_i(R(x_j))$ is either true or undefined. Using Theorem 6 we can find an input Z that causes P to recurse k times before exiting by output function i. Let Y be the union of X and Z. Since $Y \geq Z$, P must recurse at least as much on Y as it did on Z. Since the final point tested is still atomic P(Y) will recurse k times before exiting by output function i. Since

$$f_i(R^\lambda(X),R^\lambda(Y)) \neq f_i(R^k(X),C^k(Y))$$

we have that $F_X \neq F_Y$.

The second case arises when $f_i$ is a constant function. By assumption 3 there is at least one output function that is not a constant function. Let $f_i$ be this function. Let the predicate $p_i$ test variable $x_j$. We can apply the same argument as before, except that it may happen by chance that P(Y) = P(X), i.e. P(Y) returns the constant value. In this case increment k by 1 and perform the same process and it cannot happen again that P(Y) = P(X). []

**Theorem 8:** If P touches a location a, then we can construct two inputs X and Y with the properties that P(X) and P(Y) are defined. Then for any Q in $\phi$, if P(X) = Q(X) and P(Y) = Q(Y), then Q must touch a.

**Proof:** Let Z be the minimal a point. Using Theorem 7 we can construct an input X such that P(X) is defined, X $\lambda$ Z, and $F_X \neq F_Z$. Let Y be X pruned at a.

We first claim that P(Y) is defined and $F_Y = F_Z$. To see this, note that every point that was tested by P in computing P(Z) and found to be non-atomic is also non-atomic in Y. Position a is atomic in both, and if the output function was defined on Z then it must be defined on Y, which is strictly larger.

Suppose that, given input Y, a program Q recurses $\lambda$ times before exiting by output function i but does not touch position a. Since X is strictly larger than Y, on X, Q must recurse at least as much and at least reach predicate i. Let the position in Y that was touched by predicate i and found to be atomic be b. Since position b is not the same as position a, position b is also atomic in X. Therefore, given input X, Q will recurse $\lambda$ and exit by output function i. But this implies by Theorem 5 that $F_X = F_Y$, a contradiction. []

**Bounding the depth of the recursion and predicate functions:** Our first set of test inputs uses the procedure given in Theorem 8 to demonstrate that each of the n primary positions in P are indeed touched.

Next, for <u>each</u> selector variable, use the procedure given in Theorem 8 to show that the first n+1 postions (by depth) must be touched. Let d be the maximum size of these m(n+1) positions. (We will assume d is at least 3 and is larger than both 2w and **olimit.**)

**Theorem 9:** If Q is a program in $\phi$ that correctly processes these 2m(n+1) points, then the recursion selectors of Q have depth d or less.

**Proof:** Consider each selector variable separately. At least one of the n+1 points touched in that variable must have been touched after Q had recursed at least once. If the recursion selector had depth greater than d, the program could not possibly have touched the point in question. []

**Theorem 10:** If Q $\varepsilon$ $\phi$ correctly processes these 2m(n+1) points, then none of the selector programs associated with the predicates can have a depth greater than d.

**Proof:** At least one of the inputs causes Q to recurse at least once; hence all the predicates must have evaluated FALSE and therefore were defined. If any of the predicates did have a depth greater than d, they would have been undefined on this input. []

Since d > **olimit** we also know that d is a bound on the output functions of Q.

We are now in a position to make a comment concerning the size of the points computed by the procedure given in Theorem 8. Let $\lambda$ be the maximum depth of the "relative root" (the current variable position relative to the original variable tree) at the time position a is touched. We know the minimal a tree is no larger than 1+w. This being the case, to find an atomic or undefined point (as in the procedure associated with Theorem 7) we will at worst have to recurse to a position 1+w deep, but no more than 1+w+d deep. Hence neither of the two points constructed in Theorem 8 need be any larger than 1+2w+d. This fact will be of use in proving Theorem 13.

**Narrowing the form of the recursion selectors:** We will say a selector function f <u>factors</u> a selector function g if g is equivalent to f composed with itself some number of times. For example, CADR factors CADADADR. We will say that f is a <u>simple factor</u> of g if f factors g and no function factors f other than f itself. Let us denote by $s_i$, $i = 1,\ldots,m$, the simple factors of $r_i$, the recursion selector functions. That is, for each variable i there is a constant $\lambda_i$ so that the recursion selector $r_i$ is $s_i$ composed with itself $\lambda_i$ times. Let q be the greatest common divisor of all the $\lambda$s. Hence the recursion selectors of P can be written as $S^q$ for some recursion selector S.

We now construct a second set of data points in the following fashion: For each selector variable $x_i$, let a be the first position touched with depth greater than $2d^2$ in $x_i$. Using Theorem 8, generate two points that demonstrate that position a must be touched. Let $T_0$ be the set containing all the (2n + 2m(n+1 + 2m) points computed so far.

**Theorem 11:** If $Q \varepsilon \phi$ computes correctly on $T_0$ then recursion selector i of Q must be a power of $s_i$.

**Proof:** Assume the recursion selector of $x_i$ in Q is not a power of $s_i$. Recall that the depth of the selector cannot be any greater than d. Once it has recursed past the depth d, it will be in a totally different subtree from the path taken by the recursion selector of P.

Since $d > 3$, it is required that Q touch a point that has depth at least 3d. Q must therefore touch this point prior to recursing to the depth d. By Theorem 9 this is impossible. []

We can, in fact, prove a slightly stronger result.

**Theorem 12:** If $Q \varepsilon \phi$ computes correctly on $T_0$ then there exists a constant r such that the recursion selectors of Q are exactly $S^r$.

**Proof:** By Theorem 11, the recursion selectors of Q must be powers of $s_i$. For each selector, construct the ratio of the power of $s_i$ in Q to that in P. Theorem 12 is equivalent to saying that all these ratios are the same. Assume they are different and let $x_i$ be the variable with the smallest ratio and $x_j$ the variable with the largest.

Let X and Y be the two inputs that demonstrate that a position a of depth greater than $2d^2$ in $x_i$ is touched. Both P and Q must recurse at least 2d times on these inputs. In comparison to what P is doing, $x_j$ gains at least one level every time Q recurses. By the time $x_i$ is within range to touch a, $x_j$ will have gone 2d levels too far. Since $2d > d + 2w$, $x_j$ will have run off the end of its input; hence Q cannot have received the correct answer on X and Y. []

Theorem 8 gave us a method to demonstrate a position is touched. We now give a way to demonstrate a position is not touched.

**Theorem 13:** If $Q \varepsilon \phi$ computes correctly on all the test points so far constructed, then for any position a not touched by P we can construct two inputs X and Y so that if P(X) = Q(X) and P(Y) = Q(Y) then Q does not touch a.

**Proof:** Let position a be in variable $x_i$. Let m be the smallest number such that after recursing m times the recursion selector i is deeper than a. Let $\lambda$ be the maximum depth of any recursion selectors at this point. Let X be the complete tree of depth 1+2d pruned at a.

There are two cases: If P(X) is not defined, assume Q touches a. The relative roots of Q cannot be deeper than 1+d at the time when a is touched. Hence the minimal a point is no deeper than 1+2d. Since X is strictly larger than the minimal a point we know that Q(X) must be defined, which contradicts the fact that Q(X) = P(X).

The second case arises if P(X) is defined. Using Theorem 7 we construct an input $Z \geq X$ such that $F_X \neq F_Z$. Let Y be Z pruned at a. Assume Q touches a. Since $Y \geq X$, Q(Y) must be defined, so assume P(Y) is defined. By construction $F_Y = F_Z \neq F_X$. But since Q touched a, $F_X = F_Y$, which is a contradiction. []

**Recursion selectors must be the same as P:** If $Q \varepsilon \phi$ executes correctly on $T_0$, then by Theorem 12, the recursion selectors of Q must be S for some constant r. From Theorem 9 we know the depth of S is no larger than d; hence there are at most d/(depth of S) choices. For each possible r (not equal to q), construct a mutant program P', which is equal to P in all respects but the mutant selectors, which are $S^r$.

In this section we will consider test cases as pairs of inputs, generated using the procedure given in Theorem 12, which return either the value YES, saying they were generated by the same straight-line program, or the value NO, saying they weren't. Other than this we will not be concerned with the output of the mutants.

If each mutant touches a point that P does not, then construct two points (using Theorem 13) to demonstrate this. If any mutant touches only points that P itself touches, then we will say P cannot be shown correct by this testing method. Call this set of test cases $T_1$.

**Theorem 14:** If $Q \varepsilon \phi$ executes correctly on $T_0$ and $T_1$, then the recursion selectors of Q must be exactly $S^q$.

Proof: Assume not, and that the recursion selectors are $S^r$ for some constant $r \neq q$. No matter what the primary positions of Q are, we know it must touch at some point the primary positions of P. It therefore must always touch the primary positions of P relative to the position it has recursed to. But, therefore, it must at least touch the points that the mutant associated with r does. []

**Testing the primary positions of P:** Consider each primary position separately. Assume that in some program Q in $\phi$ the position is not primary, but that it is touched after having recursed $\lambda$ times. Let b be the position of a relative to $S^{q\lambda}$. This means in Q that b is primary. Now b cannot even be touched (let alone be primary) in P because of the assumption of freedom. Using the procedure given in Theorem 13, construct two points that demonstrate that b is not touched, which demonstrates that a must be primary. Taken together, these test points insure that the primary positions of P must be primary in all other programs.

Notice that we need to make no other assumptions about the other primary positions in Q; we can treat each of them independently. We, therefore, have at most n(d/(depth of $S^q$) mutant programs, hence at most twice this number of test points. Call this test set $T_2$.

**Theorem 15:** If $Q \varepsilon \phi$ executes correctly on $T_0$, $T_1$, and $T_2$ then the primary positions of Q are exactly those of P.

Notice that by Theorem 5 this also gives us the following.

**Theorem 16:** The output functions of Q are exactly those of P.

**Main Theorem:** Once we have the other elements fixed, the constructors are almost given to us. Remember one of the assumptions is that each of the constructor variables appears in its entirety in at least one of the output functions. All we need do is to construct P data points so that data point i causes the program P to recurse once and exit using an output function that contains the constructor variable i. Call this set $T_3$. Using Theorem 5 we then have

**Theorem 17:** The recursion constructors of Q must be exactly those of P.

The only remaining source of variation is the order in which the primary positions are tested. The only solution we have been able to find here (short of making more severe restrictions on $\phi$ ) is to try all possibilities. There are n! of these, some of which may be equivalent to the original program. Let $T_4$ be a set of data points that differentiates P from all non-equivalent members of this set.

Putting all of this together gives us our main theorem:

**Theorem 18:** Given a program P in $\phi$, if Q $\varepsilon$ $\phi$ executes correctly on the test points constructed in Theorems 9, 14, 15, and 17, then Q must be equivalent to P.

**Corollary:** Either P is correct or no program in $\phi$ realizes the intended function.

Even though the depth of the output functions is bounded, we did not bound the number of CONS functions they contain; hence there are an infinite number of programs in the set $\phi$. This is true even after we have bounded the depth of the recursion selectors and the predicate selectors in Theorem 10.

The most important aspect of this result is the method of the proof. Once we have fixed the recursion selectors via test set $T_0$, the remainder of the arguments can be proved by constructing a small set of mutants and showing that test data designed to distinguish these from the original actually will distinguish P from a much larger class of programs. In all we constructed

$$d(1/(\text{depth of } S) + n/(\text{depth of } S^q)) + p + n!$$

mutants, and we proved that test data that distinguished P from this set of mutants actually distinguished P from the infinite set of programs in $\phi$.

## Bibliographic Notes

The results in this chapter were developed in Budd's thesis [Budd, 1980] and in papers by Budd and Lipton [Budd, 1978] and Budd, DeMillo, Lipton and Sayward [Budd, 1980b].

# Chapter 4.

## A Mutation Analyzer

In overall structure, a mutation analyzer serves as a test harness and aids in performing mutation analysis. This chapter provides a detailed description of the implementation of a mutation analyzer.

Although existing mutation analyzers differ in certain respects, there are essential similarities. Briefly, the systems allow an interactive user to enter a program to be tested. The program is parsed to a convenient internal form and appropriate data files are created. The user then enters test data, executing the program on the test data to check for errors. At the point of calculation of the mutation score, the user "turns on" or enables a subset of the mutant operators. The system creates a list of mutant description records, descriptions of how the internal form is to be modified to create the required mutant. The changes are induced sequentially with additional heuristics to speed up processing and the modified internal form is executed. The results are compared to the original results to determine whether or not the mutant survives the execution on that data. At the completion of the pass, summary reports are presented to the user, and several options are provided for examining the remaining live mutants. The user may also declare mutants to be equivalent and therefore remove them from future consideration. This function can be partially automated with considerable improvement in performance. The issue of equivalent mutants will be discussed more fully in Chapter 8.

## System Overview

The user interface of a mutation analyzer is interactive. Tasks are assigned to both the user and the analyzer which are best suited to their capabilities. One way to see how this might be accomplished is to imagine the system as an adversary who, when confronted with a program asks the user a set of questions about the program (e.g., "Why did you use this type of statement here when an alternative statement works just as well?"). The task of the user is then to provide justification in the form of test data which will give an answer to such a question.

An overview of the structure of such a system is shown in Figure 1.

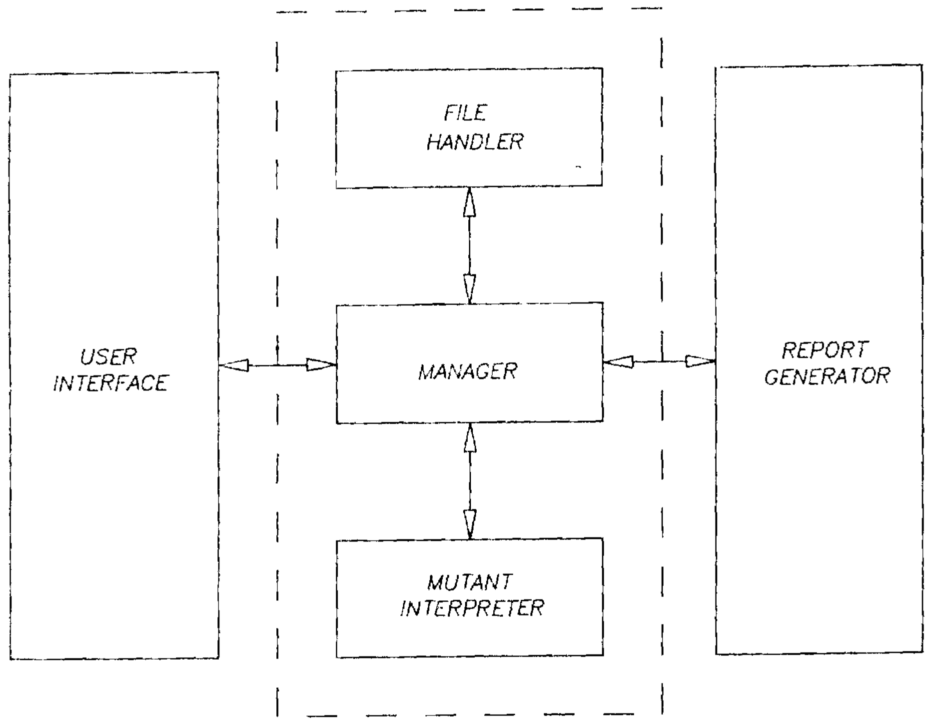



Figure 1.
System Organization

The heart of a mutation analyzer is roughly that portion of the system which lies within the dotted box in Figure 1. This portion is largely language independent since it is driven by an <u>internal form</u> of the source program rather than the source program itself. Given a sufficiently general internal form, it is possible to implement a mutation analyzer for a new language by modification of the input/output interface. In later sections, we will describe the details of a mutation analyzer for a simple subset of Cobol.

A single run of a mutation analyzer divides naturally into three phases: the <u>run preparation</u> phase, in which the information which is required by the analyzer is prepared, the <u>mutation</u> phase, during which the mutations are generated and a mutation score is calculated, and a <u>post run</u> phase in which results are analyzed and reports are generated.

**Run Preparation.** The role of the run preparation phase is to initialize various files and buffer areas. This phase is characterized by its high degree of user involvement. The user is first asked to supply the name of the file which contains the source program to be tested. Depending on whether or not the system has previously been run on this file the program file is either parsed to an internal form or a previously generated internal form file is retrieved. This internal form is subsequently interpreted to simulate program execution. A fragment of a typical internal form generated by the Fortran statement

```
IF (A .LT. X(2)) P = 1
```

is shown in Figure 2.

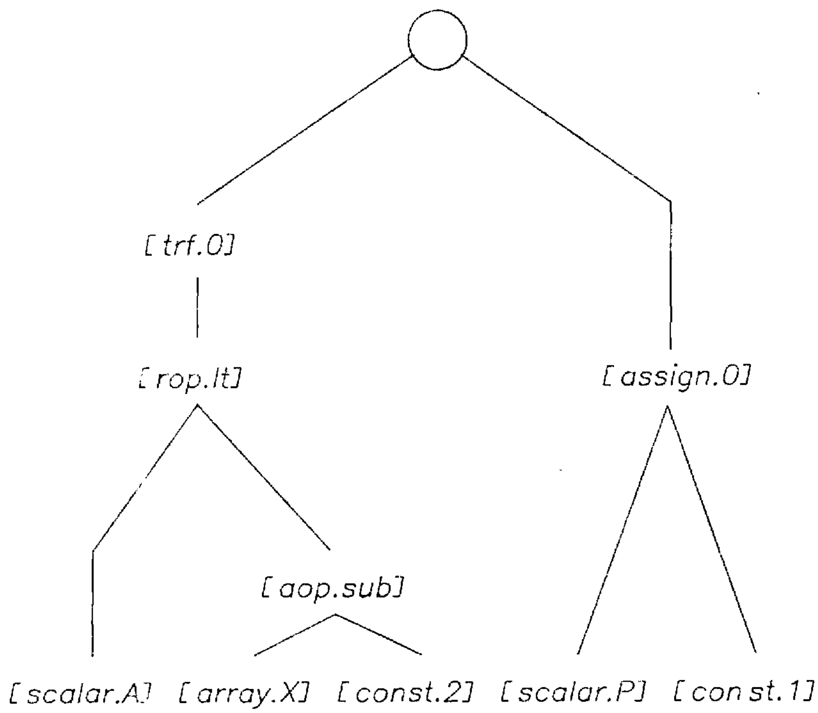



Figure 2.
Internal Form

The user is then interactively prompted for the test data on which the program is to be tested (and against which the mutation score is to be calculated). After each test case has been specified (either by direct user entry at the keyboard or by reference to a test file), the original program is executed on the test case and the results of execution are displayed (or written onto an output file for later examination). The role of the oracle who determines whether or not the calculated output of the program is satisfactory may be played by either the user or the system. If the user plays the role of the oracle, then he must literally examine the input-output relation determined by the program's execution to determine whether the computed input-output relation is the one required by the specification. If the system plays the role of the oracle, it must be supplied with a predicate subroutine. A predicate subroutine is an executable, uniform specification of input-output behavior, The system invokes the predicate subroutine each time the subject program is executed on a test case to determine if the input-output relation computed during that execution is the one required by the specification. In either case, if the test case is processed satisfactorily, the user is allowed to either enter additional test cases or to compute the mutation score and associated statistics.

After the user has entered test data, he is prompted for a specification of which mutant operators he wishes to apply. Instead of constructing multiple copies of the program (one for each mutant), a short descriptor of each mutation to be performed is generated and stored in an auxiliary file. Each time the mutant is to be run, the internal form is modified according to the information stored in the descriptor and the modified program is interpreted in the mutation phase. The user may also specify a percentage of the mutant operators to be applied.

Experience has shown that it is best to partition the task of developing test data which is adequate relative to the entire set of mutants in stages. Each stage further refines the test data to distinguish the program under test from a more extensive class of mutants. A convenient partitioning of the mutant operators is the following:

Level 1: Statement Analysis

Goal: Insure that every branch is taken and that every statement is necessary

Mutants: all statement and control mutants

Level 2: Predicate Analysis

Goal: Exercise predicate boundaries

Mutants: Alter predicate and loop limit subexpressions by small amounts

ABS insertions in predicates

Relational operator substitutions

Level 3: Domain Analsysis

Goal: Exercise data domains

Mutants: Alter constants and subexpressions by small amounts

ABS insertions

Level 4: Coincidental Correctness Analysis

Goal: Determine coincidental correctness conditions

Mutants: Operand substitutions

Operator substitutions.

In addition, the user may specify that certain of the mutants are to be randomly sampled in computing the mutation score. While there is some loss of effectiveness in randomly sampling mutants (as opposed to exhaustively executing all mutants), experimental evidence (cf. Chapter 5) suggests that test data which delivers a high mutation score under the sampling strategy also results in a high mutation score when computed according to the definitions in Chapter 1. The advantage to the user in reducing processing time can be considerable, especially for large monolithic programs.

**Mutation Phase.** Once the user has specified the program, test data and level of test (mutation operators and percentage) to be applied, the system enters the mutation phase. During this phase there is virtually no user interaction. Mutation descriptor records are processed sequentially or randomly sampled depending on whether or not the user has specified a percentage other than 100%. The mutant program is generated by modification to the internal form of the source program. The mutant is then executed on the test data and is either marked "dead" or "alive". A mutant is marked dead if it has delivered results which differ from the program being tested -- by, for example, producing different output, violating a predicate subroutine, or inducing a runtime error — on at least one test case. Otherwise the mutant remains alive. The mutation score is then the ratio of dead mutants to the total number of nonequivalent mutants. A dynamic record is kept of the number and percentage of living mutants of each type. These records are organized to allow access in a number of dimensions (e.g., live mutants by statement, by mutant type, randomly sampled). Since the final mutation score is the ratio of dead mutants to the total number of nonequivalent mutants, equivalent mutants must be deleted before the score is correctly interpretable. There are two times when it is appropriate to delete equivalent mutants. Many equivalent mutants can be detected automatically (cf. Chapter 8).

If a mutant can be deleted automatically it is deleted during the mutation phase. Equivalent mutants can also be deleted under user control during the post run phase.

**Post Run Phase.** When the mutant programs have been run on the current test cases, the system enters a post run phase. In this phase, statistics are displayed indicating the results of the mutation run to that point. The user can interactively select descriptions of live and dead mutants and display them on the screen. During the post run phase certain reports may also be generated; these reports provide a detailed permanent record of the mutation run.

The user may also declare certain mutants to be equivalent. Equivalent mutants do not enter into the mutation score calculation. There are two reason a user may declare a mutant to be equivalent. First, the user may have actually determined that the mutant belongs to $\mu_E$. Such a mutant has not been automatically eliminated during the mutation phase, but the system provides some automated help in the post run phase for determining equivalence. Some implementations provide data flow analyzers and various static analysis tools that allow the user to determine equivalence (see Chapter 8) Second, the user may choose to ignore a portion of the program being tested. For example, a subroutine or module may already have been tested adequately during a previous phase. The decision to mark all mutants which change code in that subroutine then essentially eliminates that portion of the program from further consideration even though the routine is still present in executable form and delivers results to modules which invoke it during the mutation and pre run phases.

The user can re-run the system and augment the test cases in an attempt to improve the mutation score. The user may also specify that additional mutation operators are to be applied to the program. This cycle can continue until the user is satisfied that the current test data is adequate relative to the given set of mutation operators.

Several files hold information between system runs. These are shown in Figure 3, which outlines the functions of each phase. The internal form file stores the parsed version of the source program being tested. The test data file stores for each test case the test data input and the results of execution of the program being tested on the test data. The mutation information file sorts the mutation descriptor records and other statistics generated during the mutation and post run phases.

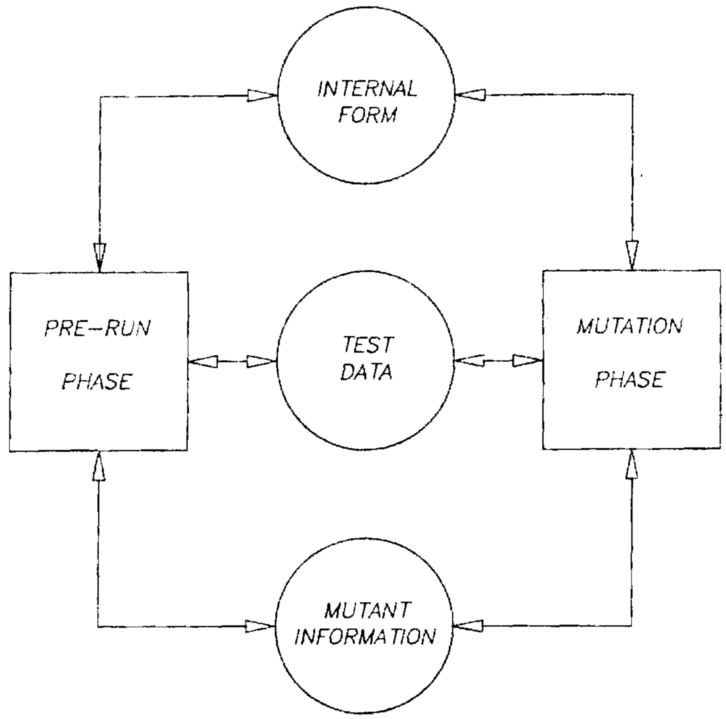



Figure 3.
Major Files

**A Mutation Analyzer for Cobol**

We will now describe in some detail the organization of a mutation analyzer for a subset of Cobol which we refer to as "Level 1" Cobol. A Level 1 Cobol program is written in the standard Cobol format (columns 1-6 containing sequence numbers, column 7 containing continuation marks, columns 8 through 72 containing Level 1 Cobol statements).

The following syntax chart defines Level 1 Cobol:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name
[AUTHOR.comment-entry.]
[DATE-WRITTEN.  comment-entry.]
[DATE-COMPILED. comment-entry.]
[SECURITY. comment-entry.]
[REMARKS. comment-entry.]

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
[SOURCE-COMPUTER.comment-entry.]
[OBJECT-COMPUTER. comment-entry.]
[SPECIAL-NAMES. ][C01 IS mnemonic-name.]

INPUT-OUTPUT SECTION.
FILE-CONTROL.
    [SELECT file-name ASSIGN TO {INPUTi|OUTPUTi}...]

DATA DIVISION.
FILE SECTION.
[FD file-name  RECORD CONTAINS integer CHARACTERS]
    [LABEL RECORDS ARE {STANDARD|OMITTED}]
    DATA RECORD IS data-name
    level-number {data-name | FILLER}
    [REDEFINES data-name-2]
    [{PICTURE|PIC} IS character-string]
    [OCCURS integer TIMES]
    ...
    ...
[WORKING STORAGE SECTION.
    [77 level entries.]
    [record entries.]...]

PROCEDURE  DIVISION.
[paragraph-name.]
    ADD {identifier-1|literal-1}[identifier-2||1-2]... {TO|GIVING} identifier-m
    [ROUNDED][ON SIZE ERROR imperative-statement].
    CLOSE file-name-1 [file-name-2]... .
    COMPUTE id [ROUNDED] = arithemtic-expression
    [ON SIZE ERROR imperative-statement ]
    DIVIDE {identifier-1|literal-1} {INTO|BY}} {identifier-2|literal-2}
    [GIVING identifier-3][ROUNDED][ON SIZE ERROR imperative-statement].
    EXIT.
    GO TO paragraph-name
    GO TO paragraph-name-1 [[paragraph-name-2]... DEPENDING ON id].
    IF condition { statement-1|NEXT STATEMENTS}
       [ELSE  statement-2 {|NEXT STATEMENT}]
    MOVE identifier-1 TO identifier-2 [identifier-3] ... .
    MULTIPLY  {identifier-1|literal-1} BY {identifier-2|1-2}
    [GIVING identifier-3][ROUNDED][ON SIZE ERROR imperative-statement].
    OPEN [INPUT file-name-1 [file-name-2]]
    [OUTPUT file-name-3 [file-name-4]]
    PERFORM  paragraph-name-1[THRU paragraph-name-2]
    PERFORM paragraph-name-1 [THRU paragraph-name -2]  {identifier-1| int-1} TIMES
```

```
PERFORM paragraph-name-1 [THRU paragraph-name-2]
  [VARYING identifier-1 FROM {identifier-2|literal-1}
     BY {identifier-3|literal-2} UNTIL condition]
READ file-name RECORD [INTO identfier]
   AT END imperative-statement
STOP RUN
SUBTRACT {identifier-1|literal-1}[identifier-2|literal-2]...
  FROM   {identifier-m|literal-m}
  [GIVING identifier-n][ROUND][ON SIZE ERROR imperative-statement].
WRITE record-name [FROM identifier-1]
  [AFTER ADVANCING {identifier-2|integer|mnemonic} LINES].
```

**Implementation Overview.** The user provides the name of the file containing the source program. Of course this program should be a legal Level 1 Cobol program. The program is parsed to its internal form. The system then produces all mutation descriptors. The legal mutations are the following:

**Decimal Alteration:** move implied decimal in numeric items one place to the left or right, if possible.

**Dimensions:** reverse two-level table dimensions

**OCCURS clause alteration:** add or subtract a constant (usually 1) from an occurs clause.

**Insert FILLER:** insert a FILLER of length 1 between adjacent items of a record.

**FILLER size alteration:** add or subtract a constant (usually 1) from the length of a FILLER.

**Elementary item reversal:** reverse adjacent elementary items in a record.

**File reference alteration:** interchange names of files at the point of reference.

**Statement deletion:** replace a statement by the null statement.

**GO TO —> PERFORM:** change GOTOs to PERFORMS

**PERFORM —> GOTO:** change PERFORMs to GOTOs

**Conditional reversal:** negate the condition in an IF-THEN clause.

**STOP statement substitution:** replace a statement by a STOP statement.

**THRU clause extension:** expand the scope of the THRU clause by a fixed number of statements (usually 1)

**TRAP statement replacement:** replace each statement by a statement. TRAP statements are not included in Level 1 Cobol. The effect of a TRAP statement is to call a routine which ceases normal program operation and returns control to the mutation analyzer with the information that a statement has been TRAPped.

**Substitute arithmetic verb:** interchange arithmetic verb with all other arithmetic verbs.

**Substitute operator in COMPUTE:** interchange arithmetic operator with all other arithmetic operators in an arithmetic expression.

**Parenthesis alteration:** move one parenthesis one character to the right or left.

**ROUNDED alteration:** interchange ROUNDED and truncation.

**MOVE reversal:** reverse the sense of a move in a simple MOVE statement if the resulting statement is legal.

**Logical operator replacement:** interchange all Boolean operators.

**Scalar for scalar replacement:** substitute one tablular item reference for another when the result is a legal expression in Level 1 Cobol.

**Constant for constant replacement:** interchange constants that appear in the program.

**Scalar for constant replacement:** replace constant references with non-tabular item references.

**Constant for Scalar replacement:** replace non-tabular item references with constant.

**Constant adjustment:** adjust the value of a constant by a fixed percentage (always at least 1 if the constant is an integer).

Mutants may be enabled selectively and a fixed precentage of the mutants to be processed may be specified as described in the previous section.

Mutants may die in a variety of ways. A mutant may deliver incorrect results (i.e., it may fail to match the output of the program being tested or may fail to satisfy the predicate subroutine). Mutants may also die by producing runtime faults (e.g., attempting to read unopened files or dividing by 0). Infinite loops in mutants are detected by setting a timing <u>constant</u> which sets an absolute upper bound on the number of iterations of a single loop which are allowed. A typical setting of the timing constant might be three times the number of statements executed by the program being tested of the test case currently being processed.

Level 1 Cobol is limited to a fixed number of sequential input and output files. Ten nonrewindable files seem to be sufficient for such common data processing applications as posting sorted transactions against a master file and updating the master. For this simple system there should be a limit set on the amount of storage allocated for each file for each test case. Files are packed into arrays by replacing each string of repetitions of a single character (such as a string of blanks) by storing a token which represents the character and a repeat count.

As described in the previous section, the system should create a number of auxiliary files. Some of these files are random access files used to process the mutants and test cases. Others are needed for the restart capability. A convenient naming scheme is to use the name of the auxiliary file as an extension to the name of the program file provided by the user. For example, if the user submits TEST-PROG-1 to the system, the system might store the internal form of the program in the file TEST-PROG-1.if.

A file that deserves special attention is the logfile. This file contains:

1. a listing of the program with line numbers assigned.

2. a record of the percentage of mutants to be created.

3. a summary of test case and mutant transactions, in the order in which they occurred (whenever a test case is submitted a message is logged about that transaction, including the location of the test case and whether the test case was accepted or rejected by the user; mutants are entered as they are enabled),

4. a summary of mutant status after each mutation phase,

5. a listing of live mutants after each mutation phase,

6. an optional listing of test cases after each pre run phase.

These files should not be automatically deleted after a run is completed, but rather should be available for a possible resumption of testing.

**Suggested File Formats.** The files which are required for processing have been described above. In this section, we will examine the structure of those files in enough detail to permit easy implementation of an analyzer for Level 1 Cobol.

**SOURCE PROGRAM <filename>**

The source program is assumed to be in a sequential system file, in the standard Cobol format.

**INPUT FILE (EXTERNAL)**

Input file can either be supplied by the user as a standard sequential file or can be entered directly from the terminal. It is, of course, possible to create some input files outside the system using whatever tools the user has access to, and to create the others interactively.

**TEST FILES (INTERNAL)**

The internal test files contain all test cases that have been created at that time. There are two files containing test information, the test status file, and the test data file.

**TEST STATUS FILE (<filename>.ts):** The first record of this file contains global information.

| entry | contents |
|---|---|
| 1 | 1 if INPUT0 is used in the program 0 otherwise. |
| 2-20 | similar for INPUT1 to INPUT9 and OUTPUT0 to OUTPUT9. |
| 21 | The total number of test cases that have been defined. |
| 22 | The number of test cases that were defined prior to this pass. |
| 23 | pointer to the next record position after the last, for appending. |

Table 1. Test Status Global Information

This record will be followed by two records for each test case. The first test case record has the format:

| entry | contents |
|---|---|
| 1 | The starting position of INPUT0 in <filename>.TD (see below) |
| 2 | The number of records in INPUT0. |
| 3-40 | Similar for the other files. |
| 41 | The number of statements executed by the original program on this testcase |

Table 2. Test Status File - Test Case Record 1

The second record contains a bit map for the statements executed by this test case. This bit map is used to speed up processing during the mutation phase. If a statement is not executed by a test case, then no mutant of that statement should be executed. By using the bit map to record statement executions, the applicability of a mutant to a given test case can be easily determined.

**TEST DATA FILE (<filename>.td):** The test data file contains the actual test cases, with the input file(s) first, followed by the output file(s) of the original program. To save space these should be stored in packed format with strings of repeated characters replaced by single characters and repeat counts.

**MUTANT RECORD FILE (<filename>.mr):** The mutant records are stored in binary format, at four integers per mutant record. All records for a particular mutant type are stored contiguously, followed by all records for the next mutant type.

**MUTANT STATUS FILE (<filename>.ms):** The first section of the file contains a total mutant count and headers for each mutant type.

| entry | contents |
|---|---|
| 1 | mutant type |
| 2 | on or off ever (initially zero) |
| 3 | on or off this run |
| 4 | mutant status file record pointer for status block |

Table 3. Mutant Status File Headers

For each mutant type there is then a status block, of one record. The status block contains the following information:

| entry | contents |
|---|---|
| 1 | total mutants for this type |
| 2 | bit map length in words |
| 3 | mrf pointer for the first mutant record of this type |
| 4 | number of live mutants |
| 5 | number of dead mutants |
| 6 | number killed by trap(*) |
| 7 | number killed by time-out |
| 8 | number killed by data fault |
| 9 | number killed by initialization fault |
| 10 | number killed by I/O fault in OPEN/CLOSE |
| 11 | number killed by attempt to read past EOF |
| 12 | number killed by writing too much |
| 13 | number killed by output too large for buffer |
| 14 | number killed by array subscripts out-of-bounds |
| 15 | number killed by incorrect output |
| 16 | number killed by garbage in the code array |

Table 4. Status Block

The status block is followed by counts indicating live, dead, and equivalent mutants, indexed by mutant number.

**INTERNAL FORM (<filename>.if):** The internal form file contains the following tables:

SYMBOL TABLE

STATEMENT TABLE

CODE ARRAY

INIT

HASH TABLE

INIT is the initial segment of memory containing literals, PICTUREs, and memory initialization information. The remaining tables are described below.

**OUTPUT FILE (<filename>.lo):** This is a file containing information on the run. Its contents are controlled by the user. Typical contents would be a listing of the source program, the test cases, the status after each pass through the system, and a listing of some or all of the live mutants.

**INITIAL.HASH:** This table is the same as HASH-TABLE except that it contains only the reserved words and their tokens.

## Internal Form Specifications

**SYMBOL TABLE:** The symbol table is an 10xN array of integers. A simple data item (group or elementary) is described by one row in the array. A table item is described in two rows, the second is a dope vector. The following conventions are useful. Entry 1 in each row (record) points to the hash table entry for the name of the item. If the item has no name (such as a filler or literal), entry 1 is zero. Entry 2 is always a code for the type of the record. Its value determines the meaning of the other entries. The overall organization of the symbol table entries is as shown in Figure 4.

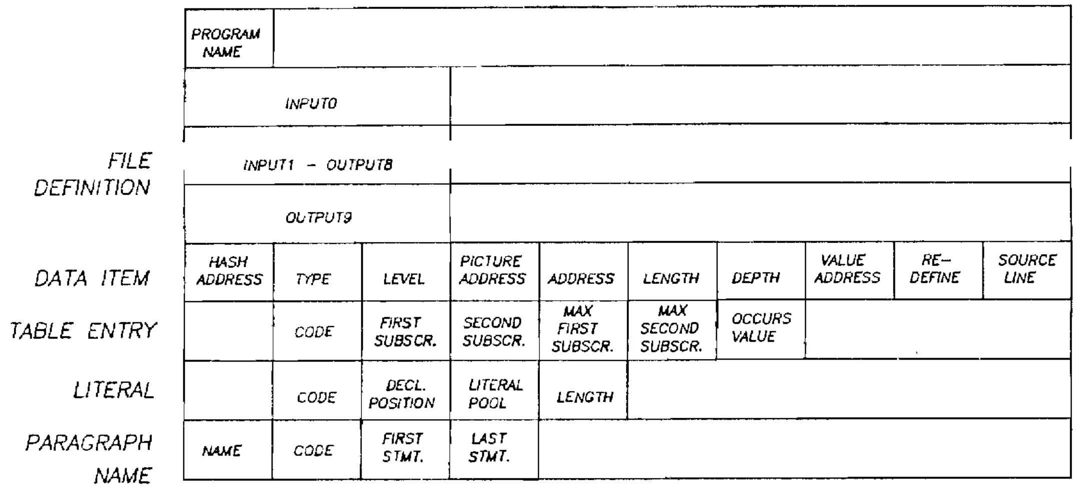



Figure 4.
Symbol Table Organization

Table 5 describes the contents of the first 21 rows of the symbol table.

| Row | Purpose | Entry | Contents |
|---|---|---|---|
| 1 | Program Name | 1 | pointer to program name |
| 2 | INPUT0 | 1 | hash table pointer to file name |
| 2 | INPUT0 | 2 | pointer to symbol table entry for data record |
| 2 | INPUT0 | 3 | record length |
| | | ... | |
| 21 | OUTPUT9 | 1 | hash table pointer to file name |
| 21 | OUTPUT9 | 2 | pointer to symbol table entry for data record |
| 21 | OUTPUT9 | 3 | record length |

Table 5. First Rows of Symbol Table

**DATA ITEMS:** The following table describes the organization of the entries for the elementary data items.

| entry | contents |
|---|---|
| 1 | Index of the identifier in the hash table, so that print name can be recalled. For FILLERS, this is zero. |
| 2 | A code for the type of the object.<br>1 for unsigned numeric identifier<br>2 for signed numeric identifier<br>3 for non-numeric identifier<br>4 for edited numeric item<br>5 for group item |
| 3 | The level number |
| 4 | Pointer to the PICTURE string in program memory for edited numeric items.<br>OR the decimal position (from right) for unedited numeric items.<br>OR not used. |
| 5 | A pointer to the start of the item in program memory. For an item in a table, this is the constant term in the address calculation. |
| 6 | The length of the item, in characters.<br>All items are stored with usage of DISPLAY. |
| 7 | The depth of the item in the table structure. (0 for scalars, 1 for one-level tables or for rows in two-level tables, 2 for two-level tables entries.) |
| 8 | Pointer to VALUE string in program memory. |
| 9 | The Symbol table row for the item that is Av REDEFINEd |
| 10 | The source program line number on which the item description began |

Table 6. Symbol Table Data Items

**SECOND ROW FOR TABLE ITEMS** A second row is required for the dope vector when the data item is a table entry.

| entry | contents |
|---|---|
| 2 | code = 6 |
| 4 | the multiplier for the first subscript. |
| 5 | the multiplier for the second subscript. |
| 6 | the maximum value for subscript-1. |
| 7 | the maximum value for subscript-2. |
| 8 | the number of OCCURances of the item. |

Table 7. Symbol Table - Table Items

**LITERALS DEFINED IN THE PROCEDURE DIVISION:** For entering references to literals which are defined in the procedure division, the following table format is used. SPACES and ZERO (and twiddles of ZERO) have entries of this format which are present by default, even if not used in the program.

| entry | contents |
|---|---|
| 2 | code = 7 for numeric literals<br>code = 8 for non-numeric literals<br>code = 10 for the twiddle of a numeric literal |
| 4 | decimal position, for numeric literal |
| 5 | pointer to value in literal pool |
| 6 | length |

Tables 8. Symbol Table - Literals

**PARAGRAPH NAMES** Paragraph names are entered in the following format:

| entry | contents |
|---|---|
| 1 | pointer to name |
| 2 | code = 9 |
| 3 | statement table index of first statement |
| 4 | statement table index of last statement |

Table 9. Symbol Table - Paragraph Names

Entries in the symbol table are stored in the same order as the items are encountered . In particular, entries for data items defined in the DATA DIVISION are stored almost as they appear in the source code, with nesting being implicit in the level numbers and the sequence. One exception to this rule is the inclusion of dummy FILLER entries of length zero between elementary items. This is to accommodate the mutant operator that inserts fillers to avoid having to change procedure division references.

Memory is organized as shown in Figure 5.

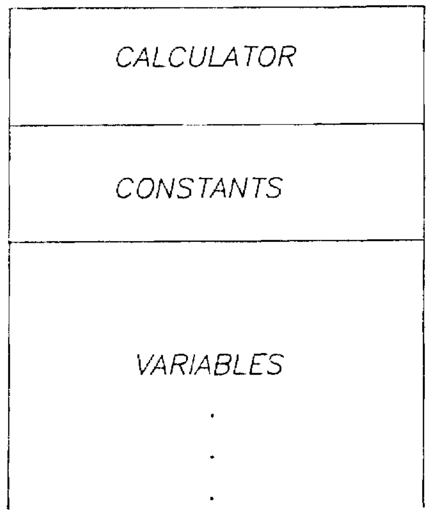



Figure 5.
Memory Organization

The first 30 characters of memory are used as a temporary arithmetic register. Following that comes the constant data area. This area includes:

**PICture strings - for edited numeric items.** There are 3+N descriptors, where N is the length of the picture string. The first is the length of the string; descriptor 2 is the number of digit positions; and descriptor 3 is the number of digits to the right of the decimal point. Then follows the picture string. An editing MOVE uses this string to interpretively execute the MOVE instruction.

**VALUE literals. for numeric items** - descriptor 1 is the number of digits, descriptor 2 is the number of digits in fraction, and descriptors 3 to n+2 are the digits themselves. An operational sign is coded in the last descriptor with the last digit. **for nonnumeric items** - descriptor 1 is the length N in characters, and descriptors 2 to N+1 are the characters.

**Procedure Division literals.** These are digits or characters only. Since these items have individual symbol table rows, the extra information (e.g., length, decimal position) is stored there.

SPACES and ZERO are stored in positions after the arithmetic register in a format that can be referenced either as VALUE or Procedure Division literals, depending on the start pointer.

A variable area follows the constant area. All data is stored on a USAGE IS DISPLAY basis, one character at a time. Since some mutations change the data structure, reallocation between executions is sometimes necessary.

**STATEMENT TABLE:** The statement table is composed of triples of integers. The first is the starting position of an instruction in the code table. When a procedure division statement is mutated, the original code is not modified. Instead, a mutated copy of the instruction is created and appended to the end of the code table. This entry is then modified to point to this mutant copy of the instruction. The second entry in the triple is the line number of the statement on the source listing. The third entry contains a code. A value of 0 means this statement is a continuation in a sentence (no period after previous statement.) A value of 1 means a new sentence. A value greater than 1 means the beginning of an ELSE clause.

**INTERNAL FORM OF PROCEDURE DIVISION:** The following table describes the format of the internal form for each Cobol instruction. The bracketed entries "identifier","ident", and "id", as well as "op" are pointers to symbol table entries describing identifiers or literals. The symbol table contains information about type, length, and location. Notice that an operand can also be a table reference. In this case, instead of a single integer we would have [op][index-1] or [op][index-1] [index-2]. The interpreter will know from the symbol table entries for op whether 0,1, or 2 indices (subscripts) are needed for a valid reference. Index-1 (and index-2) are also symbol table references to simple (unsubscripted) variables or to numeric literals. The notations "procedure" and "proc" represent pointers to symbol table entries describing paragraph names. The symbol table will contain pointers to the first and last statements in the paragraph, in the statement table.

Each instruction is preceded by a word containing the length of that instruction.

| source | internal form syntax |
|---|---|
| MOVE | <MOV><n><source><dest-1>...<dest-n> |
| ADD | <AD><rnd><size><n><op-1>...<op-n> |
| ADD-GIVING | <ADG><rnd><size><n><op-1>...<op-n><dest> |
| SUBTRACT | <SU><rnd><size><n><op-1>...<op-n> |
| SUB-GIV | <SUG><rnd><size><n><op-1>...<op-n><dest> |
| MULTIPLY | <MU><rnd><size><op-1><op-2> |
| MULT-GIV | <MUG><rnd><size><op-1><op-2><dest> |
| DIVIDE | <DI><rnd><size><op-1><op-2> |
| DIV-GIV | <DIG><rnd><size><op-1><op-2><dest> |
| COMPUTE | <CO><rnd><size><ident><arith. exp.> |
| GO TO | <GO><procedure> |
| GO TO...DEPEND | <GOD><n><proc-1>...<proc-n><ident> |
| PERFORM | <PE><procedure><procedure-2> |
| PERFORM-UNTIL | <PEU><proc-1><proc-2><condition> |
| PREFORM-VARYING | <PEV><proc-1><proc-2><ident><from><by> |
| | <REP1><p1-stmt-ptr><p2-code-ptr><condition> |
| PERFORM-TIMES | <PET><procedure><procedure-2><ident> |
| | <REP2><count><start><stop> |
| no op | <RET><0> |
| return | <RET><addr> |
| IF | <IF><else-stmt-ptr><condition> |
| NEGATED IF | <NIF><else-stmt-ptr><condition> |
| OPEN | <OP><1..20> |
| CLOSE | <CL><1..20> |
| READ | <RE><1..10><from-ident> |
| WRITE | <WR><1..10><from-ident><advance> |
| STOP RUN | <STOP> |
| TRAP | <TRAP> |

Table 10. Internal Form Syntax

The items <rnd> and <size> are codes. <rnd> is set to 0 for truncated values and 1 for rounded values. <size> is set to 0 if no SIZE ERROR clause has been specified and 1 otherwise. In the internal form the SIZE ERROR clause immediately follows the current statement. Arithmetic expressions are interpreted (see algorithms below) by a "calculator" that uses the initial memory locations for subexpression and intermediate storage.

In PERFORM-VARYING and PERFORM-TIMES statements <REP1> represents the iteration control instruction. On returning from the PERFORM, control is returned to this instruction. <p1-stmt-ptr> is a statement table pointer corresponding to the symbol table <pointer proc-1>. < p2-code-ptr> is a code pointer for the insertion of the return. <REP2> is similar to REP1, but <count> holds the value that was in <ident> when the statement was first executed. Start and stop are statement table pointers for the perform range.

Each paragraph ends with a no op statememt. When a PERFORM statement is executed, it first changes the no op at the end of its range to a return by inserting the return address (in the statement table) and then transferring to the beginning of the range. When a RETURN is executed, it transfers to the address in the instruction and also changes itself to a no op by changing its address field to 0. No op's are also inserted when NEXT SENTENCE is used or implied in an IF statement.

In the WRITE statement <advance> is a symbol table pointer.

MUTANTS: The mutant descriptions are stored in four integers. The first is the mutant type, and the others (not all types use all four integers) are used for auxiliary information. The following mutants are defined.

| mutant | semantics |
|---|---|
| DECIMAL | Move implied decimal in numeric items one place |
| DIMENS1 | Reverse row and column OCCURS counts |
| DIMENS2 | Increment or decrement (by 1) an OCCURS count. |
| INSERTF | Insert a filler with PICTURE X. |
| ALTERF | Alter a filler with PICTURE X(n) to X(n-1) or X(n+1) |
| REVERSE | Reverse adjacent elementary items in a record. |
| FILEREF | Change a file reference from one file to another |
| DELETE | Delete a statement (change it to a NO-OP). |
| GO-PERF | Change a GO TO to a PERFORM |
| PERF-GO | Change a PERFORM to a GO TO. |
| THENELS | Reverse the THEN and ELSE clauses in an IF |
| STOPINS | Insert a STOP RUN in the program. |
| THRUEXT | Extend the TRHU range of a PERFORM. |
| TRAP | Change a statement to a TRAP |
| ARIVERB | Change one arithmetic verb to another. |
| ARIOPER | Change an arithmetic operator in a COMPUTE statement. |
| PARENTH | Alter the parenthesization of an arithmetic expression |
| ROUND | Change rounding to truncation, or vice versa. |
| MOVEREV | Reverse the direction of the MOVE |
| LOGIC | Change a logical comparison to some other comparison. |
| S-FOR-S | Substitute one scalar data references |
| C-FOR-C | Substitute a constants (numeric or nonnumeric literal) |
| C-FOR-S | Substitute a constant for a scalar. |
| S-FOR-C | Substitute a scalar for a constant. |
| CONSADJ | Increment or decrement a numeric literal by 1 or by 1% |

Table 11. Mutant Semantics

We now describe the effects of each of these mutations on the internal form entries. The mutations are grouped by the Cobol syntactic structures affected during the mutation: data, input, output, control, and procedural. Each mutant is described by four integers which specify the type of mutation, relevant table entries, and parameters defining the mutant. In the notation below, blank entries in the descriptors are indicated by <x>. <field> denotes the location in the code table relative to the start of the statement. All other locations and limits are defined through their symbol table entries. Thus, the mutants can be stored in a file of 4xN integers.

**DATA MUTATIONS**

(1) <DECIMAL><sym.tab.loc><+1 | -1><x>
(2) <DIMENS1><sym.tab.loc><x><sym.tab.loc.-2>
(3) <DIMENS2><sym.tab.loc><+1 | -1><x>
(4) <INSERTF><symbol table location><x><x>
(5) <ALTERF><sym.tab.loc><+1|-1><x>
(6) <REVERSE><sym.tab.loc.><next.elementary.loc><x>

**INPUT/OUTPUT MUTATIONS**

(7) <FILEREF><statement><x><new file-code>

**CONTROL STRUCTURE MUTATIONS**

(8) <DELETE><statement><x><x>
(9) <GO-PERF><statement><x><x>
(10) <PERF-GO><statement><x><x>
(11) <THENELS><statement><x><x>
(12) <STOPINS><statement><x><x>
(13) <THRUEXT><statement><new paragraph limit><x>
(14) <TRAP><statement><x><x>

**PROCEDURAL MUTATIONS**

(15) <ARIVERB><statement><new operation><x>
(16) <ARIOPER><statement><field><new operation>
(17) <PARENTH><statement><from-field><to-field>
(18) <ROUND><statement><x><x>

(19) <MOVEREV><statement><x><x>
(20) <LOGIC><statement><field><new value>
(21) <S-FOR-S><statement><field><new symtab loc.>
(22) <C-FOR-C><statement><field><new loc>
(23) <C-FOR-S><statement><field><new loc>
(24) <S-FOR-C><statement><field><new loc>
(25) <CONSADJ><statement><field><new loc>

## Processing Algorithms

In this section, we will describe the principal processing that takes place during the mutation phase of the analyzer. The overall organization of these algorithms is as shown in Figure 6.

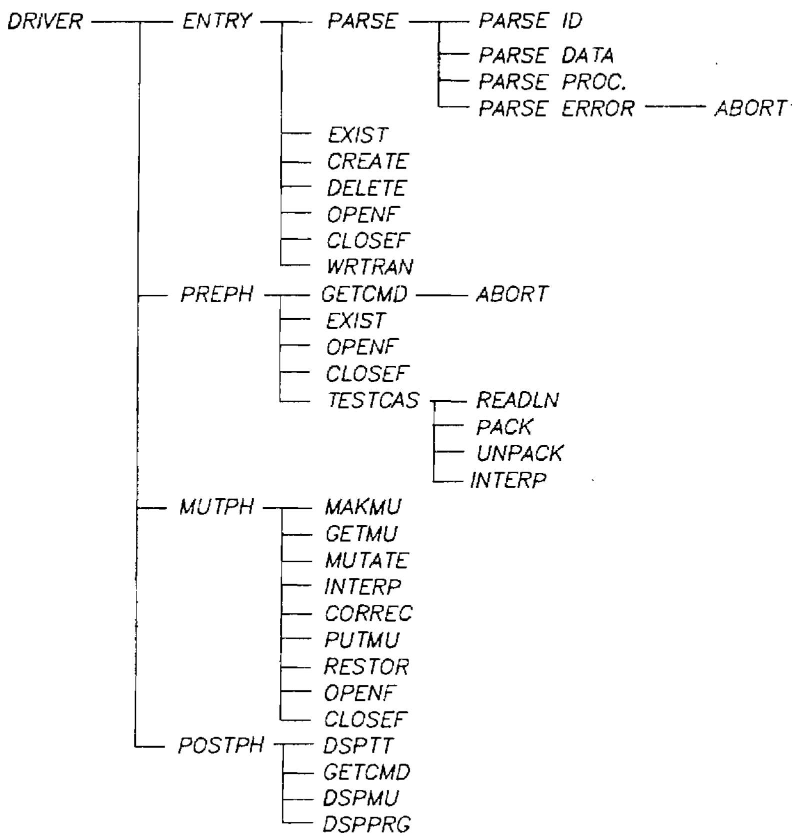



Figure 6.
Call Structure for Processing Algorithms

Each major algorithm is described below. Minor algorithms are described briefly in the major algorithms that use them.

In addition to the processing algorithms described below, an implementor will need some utilities for common file processing operations. The utilities which are most likely to be helpful are those which take and replace a given mutant (indexed by its number) in a mutant buffer, create and delete files, check to see if a specified file (on a specified unit) is open or already exists. Sequential and random access reads and writes are also required.

**ABORT - stop the run**

ABORT prints a message indicated in its call. It then closes all open files without further processing. No files are deleted. ABORT then terminates the run and returns control to the operating system. Be aware that ABORT does not actually cause the output file to be printed. The user must do that outside the system.

**ALCATE — allocate storage**

ALCATE scans the symbol table, filling in the fields for the lengths of group items, and for the positions and multipliers for all items.

**CLOSEF - close a file**

CLOSEF closes currently open files. It will also detect if the file was not opened and return an error message to the calling algorithm.

**CORREC - check mutant correctness**

CORREC compares an output of the program being executed with the output of the original program. Depending on the mode of correctness checking chosen by the user (or by the default methods), this may be done after each record is "written", after the program has completed execution (unless the program has failed by some other method), or not at all. Also selectable by the user should be the precision of the checking: total agreement, or agreement up to spacing.

**DECOMP - decompile statement**

DECOMP decompiles a statement in internal form to its Cobol equivalent.

**DRIVER the main program**

This program controls the looping through the mutation process at the highest level. It controls the prerun, mutation, and postrun phases of the run. This is the routine that may be altered later if the "phase" concept is dropped.

**DSPSTT - display status**

Display the status of the mutants that have been turned on. This includes a listing by mutant type of the numbers of mutants live and eliminated, and a listing by elimination method of the number eliminated by each method.

**ENTRY - entry routine for set-up.**

This algorithm is entered only once, at the beginning of a testing session. ENTRY first asks the user for the name of the raw program file. It then checks to see if the temporary files needed already exist (their names will be derived from the raw program file name). If they do, then the user will be asked if he wants to purge them for a fresh run. If a fresh run is desired, or if the temporary files did not exist, ENTRY causes the program to be parsed, and causes the needed temporary files to be created and initialized.

**INITM - initialize core memory**

This algorithm initializes program memory for the start of an interpretive interaction. This routine is called before each execution of each mutant program, as well as before the execution of the original program.

**INHASH — insert info into hash table**

INHASH can only be used after QHASH has already been called to determine the proper point of insertion for the name. QHASH also does the actual insertion of the name. INHASH makes the insertion permanent. If a name is not permanently inserted the name will be overwritten the next time QHASH accesses that location.

**INTERP - interpretively execute the program.**

INTERP interprets the internal form of the program. The program can fail in INTERP by attempting to read past the end of file, by writing too many records on an output file, by taking too much time, by arithmetic fault, or by mode mismatch. The limits for time and output records are in ERSTAT. For the original program these are arbitrary values, but for mutant programs, they should be set for comparisons with the original program. INTERP leaves a code for the mode of failure, or nonfailure, in ERSTAT. Also placed in ERSTAT are counts of the actual time used and records written. INTERP calls CORREC after each "write" or after the end of execution, or not at all, depending on the correctness checking mode selected by the user.

**MAKEMU - make mutants**

MAKEMU creates the descriptor record file, and initializes the mutant status record. The first time it is called, it writes header information and the first batch of mutants. On subsequent calls it appends mutant records.

**MUTATE - mutate the program**

MUTATE mutates the program. For a data division mutation, this means altering one or several entries in the symbol table, and also possibly the already initialized memory. For the procedure division the affected statement is copied, in its mutated version, at the end of the code table. The statement table is then modified so that the pertinent entry points to the modified version, rather than the original. The original statement is not affected, so that restoration is easy.

**MUTPH - control the mutation phase.**

MUTPH first creates the mutants that have been requested by the user, and then performs the mutations and runs the mutants, updating the mutant status as it does so. Each test case and each mutant record carries a flag that indicates whether or not it was created on this pass. While looping through the mutants, each new mutant is run against all test cases. Each old mutant that has not already been killed is run only against the new test data.

**OPENF - open a file**

OPENF opens a file. This algorithm will have concentrated system dependencies. Typical parameters passed to OPENF include the type of file (e.g., sequential output file or random input file), the starting position in the file (e.g., beginning, end, random address), and a flag to indicate success of the operation. Extensive use should be made of the native operating system file handling routines in implementing OPENF.

**PARSE - driver routine for parsing subroutines.**

This routine controls the four divisional routines that actually perform the parsing. It also prints error messages. The pilot system, at least, will abort the parsing when the first error is detected. The user will be informed of the offending line and the type of error.

**POSTPH — the post run phase**

POSTPH is guided by user dialogue. Its purpose is to display information for the user. The mutant status should be automatically displayed upon entry, all other information is by request. The user may ask to see the program, the test cases (by number), or the mutants (all, selected, or one random mutant of each type).

Finally, the user may return to the pre-run phase by command or end the session.

**PREPH - the controlling routine for the prerun phase**

The prerun phase is guided by user dialogue. PREPH will ask about test cases for this pass. These may be in a file or they may be entered from the terminal. Several test cases may be entered at once. After each test case the user is presented with the results of the run and is asked if the test case should be retained. After the test cases are entered PREPH asks the user which mutants are to be turned on. The user may turn them all on, or he may name a subset, or he may select mutants to be activated.

**PRSDAT - parse the data division**

PRSDAT parses the data division, building the symbol table for later use by PRSPRO, INTERP, and MAKEMU. PRSDAT enters one line in the symbol table for each identifier declared in the DATA DIVISION. PRSDAT also builds an array for the initialization of memory before each run.

**PRSENV - parse the environment division.**

This routine parses the environment division. The only lines of importance are the SELECT statements, which contain the file declarations, The file names are placed in the symbol table in entries 2-5.

**PRSID - parse the identification division.**

This routine essentially recognizes a correct identification division. The only effect on the internal form is to insert the program name (from the PROGRAM-ID statement) into the first location of the symbol table.

**PRSPRO - parse the procedure division**

RSPRO parses the procedure division, creating the code array and the statement array. PRSPRO also adds literals and paragraph names to the symbol table.

**PUTNAM - put name in NAMES array**

PUTNAM inserts character string in NAMES for future reference, such as by decompiler.

**QHASH — query hash — is item already in hash table?**

QHASH takes a name of 30 characters and checks to see if it is already in the hash table. If so, it sets and index to the position in the table where the name was found. If no match is found, an index is set to that insertion position.

**RESTOR - restore a mutant to the original version**

Restore the internal form of the program to its original state. For a Data Division mutant this means removing a filler, re-reversing two elementary items, or restoring table attributes. In all of these cases the symbol table must be modified, and space must be reallocated. For a Procedure division mutant, restoration is easier. All that must be done is to change entry 1 in the statement table entry to its previous value.

**SCAN - the scanner routine**

SCAN passes to the parsing routines tokens from the source file . For an idenfifier token, scan calls the hash query routine to see if the symbol is already in the table and if so, where.

**TSTCAS — process a test case**

TSTCAS inputs one test case from the user, either directly or from a file, runs the test case, and displays the result to the user. If the test case is accepted, it is merged into the test file, marked as "new".

## A Testing Session

The following is the output of a level 1 Cobol system whose design parallels the design given above. The program under test was modified somewhat, mainly in the reduction of the record sizes to make a better CRT display. The program takes as input two files, representing an old backup tape and a new one. The output is a summary of the changes. The input files are assumed to be sorted on a key field. The program has 1195 mutants, of which 21 are easily seen to be equivalent to the original program. Initially ten test cases were generated to eliminate all of the nonequivalent mutants. Subsequently a subset of five test cases was found to be adequate. The entire run took about 10 minutes of clock time, and 2 minutes and 13 seconds of CPU time on the PRIME 400.

```
WELCOME TO THE COBOL PILOT MUTATION SYSTEM
PLEASE ENTER THE NAME OF THE Cobol PROGRAM FILE:>log-changes
DO YOU WANT TO PURGE WORKING FILES FOR A FRESH RUN ?>yes
PARSING PROGRAM
SAVING INTERNAL FORM
WHAT PERCENTAGE OF THE SUBSTITUTION MUTANTS DO YOU WANT TO CREATE?>100
CREATING MUTANT DESCRIPTOR RECORDS
PRE-RUN PEASE
DO YOU WANT TO SUBMIT A TEST CASE ? >program

 PROGRAM LAST COMPILED ON 1 11 80.

      IDENTIFICATION DIVISION.
      PROGRAM-ID. POQAACA.
      AUTHOR. CPT R W MOREHEAD.
      INSTALLATION. HQS USACSC.
      DATE-WRITTEN. OCT 1973.
      REMARKS.
          THIS PROGRAM PRINTS OUT A LIST OF CHANGES IN THE ETF.
          ALL ETF CHANGES WERE PROCESSED PRIOR TO THIS PROGRAM.  THE
          OLD ETF AND THE NEW ETF ARE THE INPUTS.  BUT THERE IS NO
          FURTHER PROCESSING OF THE ETF HERE.  THE ONLY OUTPUT IS A
          LISTING OF THE ADDS, CHANGES, AND DELETES.  THIS PROGRAM IS
          FOR HQ USE ONLY AND HAS NO APPLICATION IN THE FIELD.
           *****************
           MODIFIED FOR TESTING UNDER CPMS BY ALLEN ACREE
           JULY, 1979.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SOURCE-COMPUTER. PRIME.
      OBJECT-COMPUTER. PRIME.
      INPUT-OUTPUT SECTION.
      FILE-CONTROL.
          SELECT OLD-ETF ASSIGN INPUT1.
          SELECT NEW-ETF ASSIGN INPUT2.
          SELECT PRNTR ASSIGN TO OUTPUT1.
      DATA DIVISION.
      FILE SECTION.
      FD  OLD-ETF
          RECORD CONTAINS 80 CHARACTERS
          LABEL RECORDS ARE STANDARD
          DATA RECORD IS OLD-REC.
      01  OLD-REC.
          03  FILLER                         PIC X.
          03  OLD-KEY                        PIC X(12).
          03  FILLER                         PIC X(67).
      FD  NEW-ETF
          RECORD CONTAINS 80 CHARACTERS
          LABEL RECORDS ARE STANDARD
          DATA RECORD IS NEW-REC.
      01  NEW-REC.
          03  FILLER                         PIC X.
          03  NEW-KEY                        PIC X(12).
          03  FILLER                         PIC X(67).
      FD  PRNTR
          RECORD CONTAINS 40 CHARACTERS
```

```
          LABEL RECORDS ARE OMITTED
          DATA RECORD IS PRNT-LINE.
      01  PRNT-LINE                                PIC X(40).
      WORKING-STORAGE SECTION.
      01  PRNT-WORK-AREA.
          03  LINE1                                PIC X(30).
          03  LINE2                                PIC X(30).
          03  LINE3                                PIC X(20).
      01  PRNT-OUT-OLD.
          03  WS-LN-1.
              05  FILLER                           PIC X VALUE SPACE.
              05  FILLER                           PIC XXXX VALUE 'O   '.
              05  LN1                              PIC X(30).
              05  FILLER                           PIC XXX VALUE SPACES.
          03  WS-LN-2.
              05  FILLER                           PIC X VALUE SPACE.
              05  FILLER                           PIC XXXX VALUE 'L   '.
              05  LN2                              PIC X(30).
              05  FILLER                           PIC XXX VALUE SPACES.
          03  WS-LN-3.
              05  FILLER                           PIC X VALUE SPACE.
              05  FILLER                           PIC XXXX VALUE 'D   '.
              05  LN3                              PIC X(20).
              05  FILLER                           PIC XXX  VALUE SPACE.
      01  PRNT-NEW-OUT.
          03  NEW-LN-1.
              05  FILLER                           PIC XXXXX VALUE ' N   '.
              05  N-LN1                            PIC X(30).
              05  FILLER                           PIC XXX VALUE SPACE.
          03  NEW-LN-2.
              05  FILLER                           PIC XXXXX VALUE ' E   '.
              05  N-LN2                            PIC X(30).
              05  FILLER                           PIC XXX VALUE SPACES.
          03  NEW-LN-3.
              05  FILLER                           PIC XXXXX VALUE ' W   '.
              05  N-LN3                            PIC X(20).
              05  FILLER                           PIC XXX VALUE SPACES.
      PROCEDURE DIVISION.
      0100-OPENS.
          OPEN INPUT OLD-ETF NEW-ETF.
          OPEN OUTPUT PRNTR.
      0110-OLD-READ.
          READ OLD-ETF AT END GO TO 0160-OLD-EOF.
      0120-NEW-READ.
          READ NEW-ETF AT END GO TO 0170-NEW-EOF.
      0130-COMPARES.
          IF OLD-KEY = NEW-KEY
              NEXT SENTENCE
          ELSE GO TO 0140-CK-ADD-DEL.
          IF OLD-REC = NEW-REC
              GO TO 0110-OLD-READ.
          MOVE OLD-REC TO PRNT-WORK-AREA.
          PERFORM 0210-OLD-WRT THRU 0210-EXIT.
          MOVE NEW-REC TO PRNT-WORK-AREA.
          PERFORM 0200-NW-WRT THRU 0200-EXIT.
          GO TO 0110-OLD-READ.
```

```
0140-CK-ADD-DEL.
    IF OLD-KEY > NEW-KEY
        MOVE NEW-REC TO PRNT-WORK-AREA
        PERFORM 0200-NW-WRT THRU 0200-EXIT
        GO TO 0120-NEW-READ
    ELSE GO TO 0150-CK-ADD-DEL.
0150-CK-ADD-DEL.
    MOVE OLD-REC TO PRNT-WORK-AREA.
    PERFORM 0210-OLD-WRT THRU 0210-EXIT.
    READ OLD-ETF AT END
        MOVE NEW-REC TO PRNT-WORK-AREA
        PERFORM 0200-NW-WRT THRU 0200-EXIT
        GO TO 0160-OLD-EOF.
    GO TO 0130-COMPARES.
0160-OLD-EOF.
    READ NEW-ETF AT END GO TO 0180-EOJ.
    MOVE NEW-REC TO PRNT-WORK-AREA.
    PERFORM 0200-NW-WRT THRU 0200-EXIT.
    GO TO 0160-OLD-EOF.
0170-NEW-EOF.
    MOVE OLD-REC TO PRNT-WORK-AREA.
    PERFORM 0210-OLD-WRT THRU 0210-EXIT.
    READ OLD-ETF AT END GO TO 0180-EOJ.
    GO TO 0170-NEW-EOF.
0180-EOJ.
    CLOSE OLD-ETF NEW-ETF PRNTR.
    STOP RUN.
0200-NW-WRT.
    MOVE LINE1 TO N-LN1.
    MOVE LINE2 TO N-LN2.
    MOVE LINE3 TO N-LN3.
    WRITE PRNT-LINE FROM NEW-LN-1 AFTER ADVANCING 2.
    WRITE PRNT-LINE FROM NEW-LN-2 AFTER ADVANCING 1.
    WRITE PRNT-LINE FROM NEW-LN-3 AFTER ADVANCING 1.
0200-EXIT.
    EXIT.
0210-OLD-WRT.
    MOVE LINE1 TO LN1.
    MOVE LINE2 TO LN2.
    MOVE LINE3 TO LN3.
    WRITE PRNT-LINE FROM WS-LN-1 AFTER ADVANCING 2.
    WRITE PRNT-LINE FROM WS-LN-2 AFTER ADVANCING 1.
    WRITE PRNT-LINE FROM WS-LN-3 AFTER ADVANCING 1.
0210-EXIT.
    EXIT.
>yes
 WHERE IS OLD-ETF?
>1c9
 WHERE IS NEW-ETF?
>1c6
 OLD-ETF PROVIDED TO THE PROGRAM

 I123456789012IIIIIIIII0JJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFF0DDDDDDDDDSSSSSSSSSXXXXXXXXXXEEEEE

 NEW-ETF PROVIDED TO THE PROGRAM
```

```
I1334567890120000000000000000000000000000000000000000000000000000000000000000000
J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEEEE
 345678901234UUUUUUUUUUHHHHHHHHHHGGGGGGGGGGDDDDDDDDDDSSSSSSSSSSEEEEEEEEEEAAAAAAA

PRNTR AS WRITTEN BY THE PROGRAM

 O    I123456789012IIIIIIIIII0JJJJJ
 L    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
 D    NNNBBBBBBBBBBGGGGGGG

 N    I13345678901200000000000000000
 E    000000000000000000000000000000
 W    00000000000000000000

 O    J234567890123YYYYYYYYYYGGGGGGG
 L    GGGFFFFFFFFFF0DDDDDDDDDSSSSSSS
 D    SSSXXXXXXXXXXEEEEEEE

 N    J234567890123YYYYYYYYYYGGGGGGG
 E    GGGFFFFFFFFFFDDDDDDDDDDSSSSSSS
 W    SSSXXXXXXXXXXEEEEEEE

 N     345678901234UUUUUUUUUUHHHHHHH
 E    HHHGGGGGGGGGGDDDDDDDDDDSSSSSSS
 W    SSSEEEEEEEEEEAAAAAAA

 THE PROGRAM TOOK  84 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >no
MUTATION PHASE
WHAT NEW MUTANT TYPES ARE TO BE CONSIDERED ?  >select

ENTER THE NUMBERS OF THE MUTANT TYPES YOU WANT TO TURN ON AT THIS TIME.

     4      ****  INSERT FILLER TYPE  ****
     5      ****  FILLER SIZE ALTERATION TYPE  ****
     6      ****  ELEMENTARY ITEM REVERSAL TYPE  ****
     7      ****  FILE REFERENCE ALTERATION TYPE  ****
     8      ****  STATEMENT DELETION TYPE  ****
    10      ****  PERFORM --> GO TO TYPE  ****
    11      ****  THEN - ELSE REVERSAL TYPE  ****
    12      ****  STOP STATEMENT SUBSTITUTION TYPE  ****
    13      ****  THRU CLAUSE EXTENSION TYPE  ****
    14      ****  TRAP STATEMENT REPLACEMENT TYPE  ****
    19      ****  MOVE REVERSAL TYPE  ****
    20      ****  LOGICAL OPERATOR REPLACEMENT TYPE  ****
    21      ****  SCALAR FOR SCALAR REPLACEMENT  ****
    22      ****  CONSTANT FOR CONSTANT REPLACEMENT  ****
    23      ****  CONSTANT FOR SCALAR REPLACEMENT  ****
    25      ****  CONSTANT ADJUSTMENT  ****

TYPES ? >4 to 14 stop
--- TESTCASE     1 ---
       250
       284 CONSIDERED        224 KILLED         60 REMAIN
```

MUTANT STATUS

| TYPE | TOTAL | LIVE | PCT | EQUIV |
|---|---|---|---|---|
| INSERT | 41 | 7 | 82.93 | 0 |
| FILLSZ | 38 | 14 | 63.16 | 0 |
| ITEMRV | 21 | 0 | 100.00 | 0 |
| FILES | 5 | 1 | 80.00 | 0 |
| DELETE | 54 | 13 | 75.93 | 0 |
| PER GO | 7 | 2 | 71.43 | 0 |
| IF REV | 3 | 1 | 66.67 | 0 |
| STOP | 53 | 10 | 81.13 | 0 |
| THRU | 8 | 2 | 75.00 | 0 |
| TRAP | 54 | 10 | 81.48 | 0 |
| TOTALS | | | | |
| | 284 | 60 | 78.87 | 0 |

```
DO YOU WANT TO SEE THE LIVE MUTANTS?>no
DO YOU WANT TO SEE THE EQUIVALENT MUTANTS?>no
WOULD YOU LIKE TO SEE THE TEST CASES?>no
LOOP OR HALT ? >loop
PRE-RUN PHASE
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>1c15
 WHERE IS NEW-ETF?
>1c5
 OLD-ETF PROVIDED TO THE PROGRAM

 0000000000012IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 I123456789012IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 NEW-ETF PROVIDED TO THE PROGRAM

 I123456789012IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 PRNTR AS WRITTEN BY THE PROGRAM

  O    0000000000012IIIIIIIIIIJJJJJJJ
  L    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  D    NNNBBBBBBBBBBGGGGGG

 THE PROGRAM TOOK  44 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>1c14
 WHERE IS NEW-ETF?
>1c5
 OLD-ETF PROVIDED TO THE PROGRAM

 I123456789012IIIIIIIIIKJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 NEW-ETF PROVIDED TO THE PROGRAM
```

```
 I123456789012IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 PRNTR AS WRITTEN BY THE PROGRAM

  O   I123456789012IIIIIIIIIIKJJJJJJ
  L   JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  D   NNNBBBBBBBBBBGGGGG

  N   I123456789012IIIIIIIIIIJJJJJJJ
  E   JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  W   NNNBBBBBBBBBBGGGGG

 THE PROGRAM TOOK  48 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>1c11
 WHERE IS NEW-ETF?
>1c1
 OLD-ETF PROVIDED TO THE PROGRAM

 000000000000000000000000000000000000000000000

 NEW-ETF PROVIDED TO THE PROGRAM

 I123456789012IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE
  345678901234UUUUUUUUUUHHHHHHHHHHGGGGGGGGGGDDDDDDDDDDSSSSSSSSSSEEEEEEEEEEAAAAA

 PRNTR AS WRITTEN BY THE PROGRAM

  O   000000000000000000000000000000
  L   000000000000000
  D

  N   I123456789012IIIIIIIIIIJJJJJJJ
  E   JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  W   NNNBBBBBBBBBBGGGGG

  N   J234567890123YYYYYYYYYYGGGGGGG
  E   GGGFFFFFFFFFFDDDDDDDDDDSSSSSSS
  W   SSSXXXXXXXXXXEEEEE

  N    345678901234UUUUUUUUUUHHHHHHH
  E   HHHGGGGGGGGGGDDDDDDDDDDSSSSSSS
  W   SSSEEEEEEEEEEAAAAA

 THE PROGRAM TOOK  64 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>1c1
 WHERE IS NEW-ETF?
>1c11
```

```
 OLD-ETF PROVIDED TO THE PROGRAM

 I123456789012IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE
  345678901234UUUUUUUUUUHHHHHHHHHHGGGGGGGGGGDDDDDDDDDDSSSSSSSSSSEEEEEEEEEEAAAAA

 NEW-ETF PROVIDED TO THE PROGRAM

 0000000000000000000000000000000000000000

 PRNTR AS WRITTEN BY THE PROGRAM

  N   00000000000000000000000000
  E   0000000000000
  W

  O   I123456789012IIIIIIIIIIJJJJJJ
  L   JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  D   NNNBBBBBBBBBBGGGGGGG

  O   J234567890123YYYYYYYYYYGGGGGGG
  L   GGGFFFFFFFFFFDDDDDDDDDDSSSSSSS
  D   SSSXXXXXXXXXXEEEEEEE

  O    345678901234UUUUUUUUUUHHHHHHH
  L   HHHGGGGGGGGGGDDDDDDDDDDSSSSSSS
  D   SSSEEEEEEEEEEAAAAAAA

 THE PROGRAM TOOK  64 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >no
MUTATION PHASE
WHAT NEW MUTANT TYPES ARE TO BE CONSIDERED ?  >all
--- TESTCASE     1 ---
      250
      500
      750
      814 CONSIDERED      640 KILLED       174 REMAIN
--- TESTCASE     2 ---
      234 CONSIDERED       82 KILLED       152 REMAIN
--- TESTCASE     3 ---
      152 CONSIDERED        1 KILLED       151 REMAIN
--- TESTCASE     4 ---
      151 CONSIDERED       61 KILLED        90 REMAIN
--- TESTCASE     5 ---
       90 CONSIDERED       69 KILLED        21 REMAIN
MUTANT STATUS

TYPE      TOTAL    LIVE     PCT   EQUIV
INSERT       41       3    92.68     0
FILLSZ       38      12    68.42     0
ITEMRV       21       0   100.00     0
FILES         5       0   100.00     0
DELETE       54       1    98.15     0
PER GO        7       0   100.00     0
IF REV        3       C   100.00     0
```

```
STOP             53          0  100.00      0
THRU              8          0  100.00      0
TRAP             54          0  100.00      0
MOVE R           13          0  100.00      0
LOGIC            15          1   93.33      0
SUBSFS          704          4   99.43      0
SUBCFC           12          0  100.00      0
SUBCFS           58          0  100.00      0
C ADJ            12          0  100.00      0

TOTALS
               1098         21   98.09      0
DO YOU WANT TO SEE THE LIVE MUTANTS?>yes
   THE LIVE MUTANTS

FOR EACH MUTANT : HIT RETURN TO CONTINUE.  TYPE 'STOP' TO STOP.
TYPE 'EQUIV' TO JUDGE THE MUTANT EQUIVALENT.

 ****  INSERT FILLER TYPE   ****

THERE ARE          3  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
 THE ITEM WHICH STARTS ON LINE 52
 ITS LEVEL NUMBER IS 3

>
 A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
 THE ITEM WHICH STARTS ON LINE 53
 ITS LEVEL NUMBER IS 3

>
 A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
 THE ITEM WHICH STARTS ON LINE 69
 ITS LEVEL NUMBER IS 3

>

 ****  FILLER SIZE ALTERATION TYPE   ****

THERE ARE         12  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 THE FILLER ON LINE 58  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 58  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 63  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 63  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 68  HAS HAD ITS SIZE  DECREMENTED  BY ONE.
```

```
>
 THE FILLER ON LINE 68  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 73  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 73  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 77  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 77  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 81  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 81  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>

 ****  STATEMENT DELETION TYPE  ****

THERE ARE          1  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 ON LINE 106  THE STATEMENT:
             GO TO 0150-CK-ADD-DEL
 HAS BEEN DELETED.

>

 ****  LOGICAL OPERATOR REPLACEMENT TYPE  ****

THERE ARE          1  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 ON LINE 102  THE STATEMENT:
             IF OLD-KEY  > NEW-KEY
 HAS BEEN CHANGED TO:
             IF OLD-KEY NOT < NEW-KEY

>

 ****  SCALAR FOR SCALAR REPLACEMENT  ****

THERE ARE          4  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 ON LINE 129  THE STATEMENT:
             MOVE LINE1 TO N-LN1
 HAS BEEN CHANGED TO:
             MOVE NEW-REC TO N-LN1

>
 ON LINE 129  THE STATEMENT:
             MOVE LINE1 TO N-LN1
```

```
 HAS BEEN CHANGED TO:
             MOVE PRNT-WORK-AREA TO N-LN1

>
 ON LINE 138  THE STATEMENT:
             MOVE LINE1 TO LN1
 HAS BEEN CHANGED TO:
             MOVE OLD-REC TO LN1

>
 ON LINE 138  THE STATEMENT:
             MOVE LINE1 TO LN1
 HAS BEEN CHANGED TO:
             MOVE PRNT-WORK-AREA TO LN1

>
DO YOU WANT TO SEE THE EQUIVALENT MUTANTS?>no
WOULD YOU LIKE TO SEE THE TEST CASES?>no
LOOP OR HALT ? >halt

**** STOP
```

## Bibliographic Notes

The paper [Acree, 1979] gives an overview of existing mutation analyzers. The basic structure described in this chapter was described in a paper by Budd, DeMillo, Lipton and Sayward [Budd, 1978a] The system described in [Budd, 1978a] accepts a subset of Fortran. Subsequent analyzers have been designed and implemented for ANSI Fortran 74 [Budd, 1980] and Level 1 Cobol [Acree, 1980], [Hanks, 1980]. Budd [Budd, 1982] has announced the implementation of a portable Fortran analyzer. Techniques for speeding up the mutation phase are described in each of these references. In addition, post processors to detect certain forms of mutant equivalence were discussed by Baldwin and Sayward [Baldwin, 1979]. Tanaka's thesis describes the implementation of an equivalence checker based on data flow analysis techniques.

# Chapter 5

## The Complexity of Program Mutation

In this chapter, we will deal with the cost of mutation analysis and with methods for reducing the cost. The efficiency of calculating the m(P,T) value for a program T is limited by the number of mutants in μ(P) and, to a lesser extent, by the running time of P. We will discuss the worst case size of μ(P) for the mutation operators described in Chapter 2 and give observed values for the size of M(P). We will also present some justification for reducing the total cost of analysis by random sampling of mutants and discuss the effects of sampling techniques on the quality of test data.

### Estimating |μ(P)|

The effects of the running time of P on the overall complexity of calculating m(P,T) are difficult to determine in quantitative terms. Because of the variety of ways in which a mutant may die, mutants tend to be very unstable. That is, a mutant may not die by actually producing an output which differs from P. It is more likely that a mutant will die by executing a trap statement, an illegal operation (a zero divide, for instance), or by one of a number of other "non-standard" means. Furthermore, not every live mutant is executed on every test case. As described in Chapter 4,it is convenient to keep a count of executed statements available during mutation phases. If a mutant occurs in an unexecuted portion of a program, then that mutant is not executed on the test case, since it cannot possibly be killed by the test case. Thus, even though programs with long running times are more costly to test by mutation analysis (or by any other dynamic testing technique, for that matter), the best estimate of the cost of calculating m(P.T) is $\mu(P)$. It is this quantity on which we will concentrate.

Mutant operators are chosen to balance two conditions. The first condition is that $\mu(P)$ be kept reasonably small -- say, a small polynomial function of some simple size parameter such as number of statements or number of data names. The second condition is that $\mu(P)$ come as close as possible to satisfying the Competent Programmer Assumption.

Recall that we have defined simple mutants as follows. Let P be a program in in a programming system defined by a grammar G, and let parse(P) be the syntax tree for P obtained by parsing P according to G. Then a 1-order simple mutant operator is a function mapping $T_1$ to a tree $T_2$ so that $T_1$ and $T_2$ differ by at most one terminal node (i.e., leaf). $T_2$ defines a simple 1-order mutant of P. Proceeding inductively, a k-order mutant is simply a k-fold iteration of 1-order mutants. In particular, notice that simple mutants do not alter the "semantic structure" of a program -- that is they do not modify the internal nodes of the parse tree. Error operators are with few exceptions simple 1-order mutants.

We will give a heuristic analysis of the expected number of mutants of a program as a function of several size parameters.

First, it is possible to derive an order-of-growth expression for the number of Fortran mutants. Data reference replacements are accomplished by interchanging reference names occurring within the program. In a program with N statements and K distinct data references this number is $F(N,K)= \binom{K}{2}=O(K^2)$. The reader can convince himself that for each of the constant and operator replacement schemes there is a constant c so that the number of generated mutants is bounded by cK. Therefore, F(N,K) dominates the total number of of mutants, and the number of generated mutants is in the worst case quadratic in the number of distinct data references.

Observations of typical programs lead to another estimation of the expected number of mutants generated. In programs that are not inordinately dense each statement contains relatively few data references, so F(N,K) is more closely approximated by $F(N,K)= O(NK)$. In typical programs, the data references tend to be so sparsely distributed that the rate of growth is usually closer to quadratic in N: $F(N,K) = O(N^2)$.

In generating mutants of Cobol programs, it is possible to more nearly approach linear growth, since the number of data reference interchanges is limited by syntactical redundancies. In fact, an analysis similar to the one carried out above gives the worst case estimate for the expected number of mutants for a Cobol program as the number of data division lines multiplied by the number of procedure division lines. For typical Cobol programs this estimate is $C(N,K) \ll N^2$.

<u>Observed</u> values of $\mu(P)$ fall considerably under these estimates. Tables 1 and 2 show mutant growth rates for some typical Fortran and Cobol programs. Notice that in both cases (except for the variation in small Fortran programs) the estimates given above are generous upper bounds on the observed number of mutants. In

experimental settings the average growth rate for "production" Cobol programs to be more nearly linear in the product of procedure division lines and K than quadratic in N.

| N | $N^2$ | Average Number of Mutants |
|---|---|---|
| 12 | 144 | 2508 |
| 13 | 169 | 307 |
| 14 | 196 | 427 |
| 16 | 256 | 360 |
| 17 | 289 | 390 |
| 24 | 576 | 2666 |
| 26 | 676 | 649 |
| 28 | 784 | 3213 |
| 30 | 900 | 1209 |
| 33 | 1089 | 12116 |
| 34 | 1156 | 3361 |
| 36 | 1296 | 1085 |
| 42 | 1764 | 1057 |
| 45 | 2025 | 1658 |
| 65 | 4225 | 1514 |
| 66 | 4356 | 2425 |
| 71 | 5041 | 2817 |
| 98 | 9604 | 8424 |
| 123 | 15129 | 8838 |

Table 1. Fortran Mutants

| N | $N^2$ | No.Procedure * No Data Div Lines | Total Mutants Generated |
|---|---|---|---|
| 57 | 3249 | 576 | 370 |
| 64 | 4096 | 789 | 679 |
| 73 | 5329 | 756 | 78 |
| 74 | 5476 | 800 | 235 |
| 75 | 5625 | 837 | 225 |
| 78 | 6084 | 918 | 376 |
| 99 | 9801 | 1674 | 377 |
| 102 | 10404 | 1806 | 715 |
| 111 | 12321 | 2115 | 740 |
| 143 | 20449 | 3330 | 628 |
| 170 | 28900 | 5184 | 1195 |
| 453 | 205209 | 46803 | 14639 |
| 670 | 448900 | 92964 | 50983 |

Table 4.2. Cobol Mutants

Choosing to measure the complexity of mutation analysis on the basis of a single size measure can, however, be deceptive. For example, consider a single assignment statement. If the right hand side of the assignment is extremely complex, then the number of data references and operators will determine completely the number of mutants generated. The 33 line program in Table 1 is an example of a program with such a dense structure.

Another size measure is the complexity of the control structure. The so-called McCabe metric measures branching complexity. The Halstead effort measurement is another measure of complexity. The following table summarizes the observed relationship between these six size measures for 16 Fortran programs.

| Number of Lines | McCabe Metric | Number Data Refs | Number Distinct Refs | Effort | Number of Mutants |
|---|---|---|---|---|---|
| N | V | X | K | E | M |
| 12 | 1 | 103 | 21 | 32033 | 2580 |
| 13 | 5 | 27 | 8 | 4071 | 317 |
| 17 | 4 | 32 | 8 | 6928 | 386 |
| 17 | 7 | 45 | 9 | 15246 | 634 |
| 24 | 7 | 72 | 40 | 17565 | 2716 |
| 26 | 9 | 40 | 11 | 16270 | 646 |
| 33 | 12 | 55 | 13 | 41819 | 859 |
| 33 | 1 | 407 | 53 | 249701 | 23382 |
| 56 | 9 | 129 | 23 | 138939 | 3657 |
| 66 | 10 | 115 | 15 | 170492 | 2425 |
| 67 | 15 | 158 | 28 | 189585 | 5230 |
| 71 | 11 | 135 | 16 | 166715 | 2888 |
| 98 | 22 | 227 | 32 | 365825 | 8457 |
| 112 | 26 | 237 | 68 | 320331 | 16380 |
| 277 | 122 | 545 | 63 | 3024488 | 34657 |
| 514 | 113 | 1138 | 93 | 19267409 | 120000 |

Table 3. Complexity Metric Relationships

The strength of the correlation of the number of mutants with each of the other measurements is given in the following table.

| Measurable Factor | Correlation Coefficient | Data Mutants | Operator Mutants | Statement Mutants |
|---|---|---|---|---|
| N | .950 | .946 | .953 | .940 |
| V | .798 | .795 | .880 | .764 |
| X | .978 | .980 | .993 | .921 |
| K | .826 | .836 | .874 | .722 |
| XK | .999 | .999 | .961 | .970 |
| E | .975 | .970 | .880 | .999 |
| M | — | .999 | .953 | .940 |

Table 4. Correlation of Complexity and Mutants

The correlation coefficient is for a linear fit between the number of mutants and the factors discussed above (first column). The second, third, and fourth columns represent the correlation between the number of mutants and the mutants arising from the three categories of mutation operators. It is possible to develop useful linear models to predict the number of mutants in terms of the most significant factors. For example, the linear model for the data above is

$$M = 79+.766XK+4X+.0008E.$$

However, this model is correlated only marginally better than the simple statistic XK. It is unlikely that the coefficients can be generalized to form a reliable predictive model for other data sets.

**Mutant Instability.**

Even though the number of mutants generated by these methods is observed to grow rather slowly as a function of program size, of the As noted above, however, a mutant seldom runs to completion; rather, mutant programs tend to be rather unstable, dying by executing "illegal" statements which are trapped and which cause premature termination of the programs. The statistics in Table 5 show typical stability data for Fortran programs tested under a mutation analyzer.

| observation | |
|---|---|
| Average number of test cases mutants remain live | 1.75 |
| .sp | |
| Average total mutant executions per session (units = F(N,K)) | 2.00 |
| .sp | |
| Average fraction of nonequivalent mutants killed by first test case | 68% |
| .sp | |
| Average execution time of live mutant (percent of original test) | 75% |

Table 5. Lifespan of Unstable Fortran Mutants

The instability of mutants has some theoretical basis. From standard software reliability studies of software we have the working principle that the probability of failure in a given time interval is proportional to the number of errors in the program. Whenever this principle holds, the expected time to failure of the program is inversely proportional to the number of errors present. If t is the time to failure (measured, say, in number of statements executed), and if cn is the probability of failure during the execution of any given statement, then the expected time to failure is given by

$$E(t) = \sum_{i=1}^{\infty} (1-cn)^{(i-1)}(cn)i \ .$$

This reduces to $E(t) = cn^{-1}$.

Although the speed with which mutants can be eliminated is a function of the capabilities of the human tester, it is our experience that somewhat more than 30% of the remaining live mutants are killed by each test case, yielding rapid convergence.

The following table represents the average number of statements executed before failure for program with k-order mutants ($k \geq 2$). The programs represented are from the set of six Cobol programs described in Appendix A.

| Program | 2nd ORder | 3rd Order | 4th Order | 5th Order |
|---|---|---|---|---|
| A1 | 30 | 24 | 21 | 19 |
| A2 | 47 | 27 | 19 | 15 |
| A3 | 50 | 38 | 31 | 27 |
| A4 | 124 | 85 | 67 | 59 |
| A5 | 52 | 35 | 27 | 22 |
| A6 | 132 | 98 | 74 | 60 |

Table 5. Time to Failure Data

As the graph in Figure 1 shows, the analytical model holds quite well. Not only is there an apparent linear relationship between 1/Avg(T) and n for each of the programs, but also for all but one of the programs, the line segments can be extrapolated backwards to show the intercepts near zero. That one program is the smallest of the six and, presumably, the worst simulation of a large module. This data cannot be interpreted as strongly as we would like, however, since the probabilistic assumptions are based on typical operational data; the test cases that generated this data were intentionally chosen to be nontypical: the test cases were required to exercise the exception-handling code that would rarely be executed in practice.

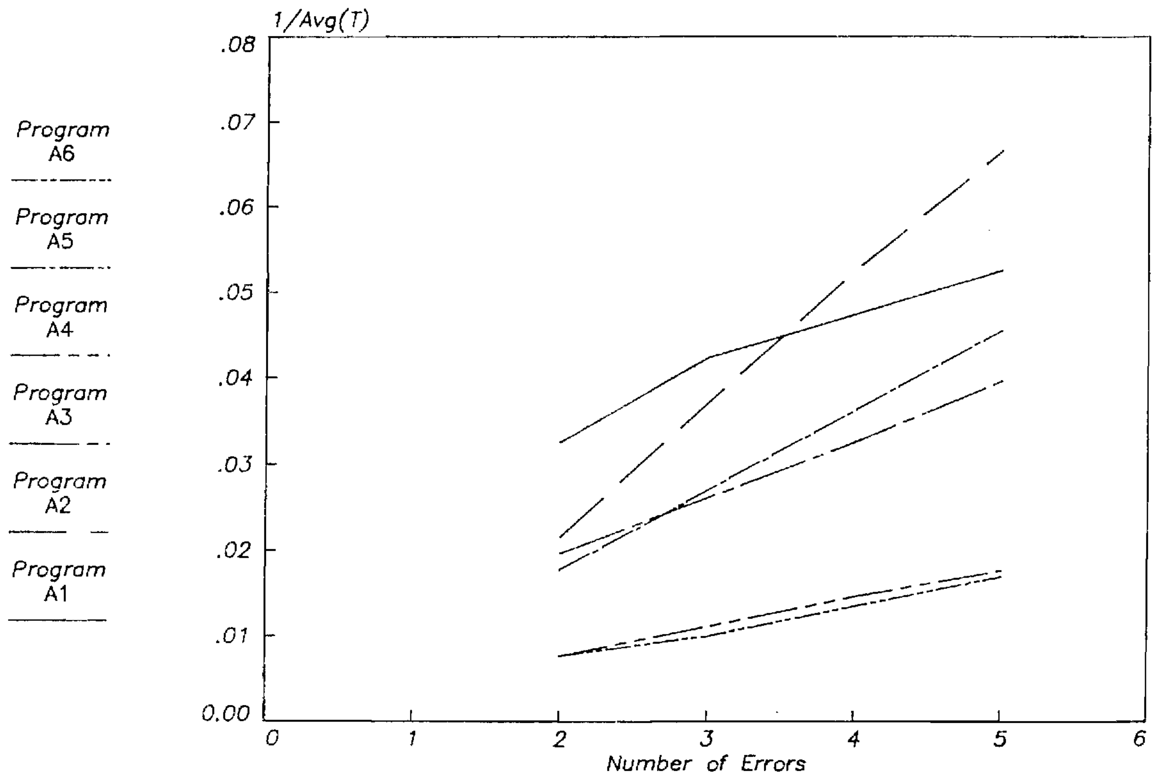


Figure 1. Failure Rate Data

### Reducing Complexity by Sampling

The bounds of practicality for monolithic programs are somewhere in the 5,000 to 10,000 line range for Fortran and somewhat higher for Cobol programs. Even this must be treated as an optimistic upper limit — certainly mutation is not easy to apply at the 5,000 statement level. A valuable technique for handling large programs is to use Monte Carlo methods to sample from large populations of mutants. A simple argument to support such an analysis goes as follows. Let f(x) appear in a specific context of a program undergoing mutation analysis; if a set of test data is too weak for the program but the program is nevertheless correct, then there is an adequate set of test data , T, on which $[f(x)]^*(T) \neq [f(x')]^*(T)$, where x' is some specified data reference replacement mutation of x and $[f(x)]^*$ denotes the functional interpretation of f(x). But x and x' in these expressions are bound variables; it only matters that they refer to distinct positions of a state vector which has been specially constructed to exhibit the inequality. In other words, it is important that we are able to "explain" with test data why x is an argument of f, but perhaps less important that we be able to explain why the argument is not x' or any other specific alternative. But this can be accomplished by sampling from enough alternative choices x' to insure that identities that we are observing are not mathematical. If the functions involved are at all well-behaved algebraically then algebraic identities can be discerned in this way

Using the Cobol program A1-A6 in Appendix A, we want to study the effects of testing using only randomly selected substitution mutants. The table which follows summarizes the results of this study. The columns labelled "survive" indicate the counts of the number of mutants (using 100% of the substitution mutants) that survive the specified testing criteria and are not equivalent to the original program.

| Program | # Mutants at 10% | # Mutants at 100% | Survive TRAP | Survive 10% |
|---|---|---|---|---|
| A1 | 389 | 1098 | 6 | 0 |
| A2 | 603 | 2814 | 906 | 0 |
| A3 | 1125 | 6340 | 129 | 2 |
| A4 | 1609 | 7334 | 97 | 16 |
| A5 | 1527 | 7957 | 407 | 14 |
| A6 | 4011 | 28275 | 789 | 66 |

Table 7. Random Sampling Experiment

We have included the strength of data that merely covers all statements for comparison purposes. While simple statement coverage does not by itself lead to strong test data, generating mutants to kill only 10% of the substitution mutants is almost as good as generating test data to kill 100% of the mutants. This trend is almost as strong at the 5% and 1% levels for large programs.

The apparent decrease in the strength of the test as program size increases is probably due to the naive sampling strategy used to sample the mutants. A sampling strategy which inserts default values or avoids selection of mutants which are correlated to previously selected mutants should avoid this effect. This experiment has been repeated several times using differing sets of programs.

In a similar experiment, three Fortran programs (B1-B3 in Appendix B) were subjected to mutation using test data that killed all nonequivalent mutants. In a double blind experiment, the same programs were analyzed by three different subjects. Subject 1 analyzed all three programs sampling 10% of the mutants, subject 2 sampled using 25% of the mutants, while subject 3 analyzed all three programs at the 50% level. The number of nonequivalent mutants left undetected by the three subjects is shown in the following table as a fraction of the total number of mutants.

| Program | 1 | 2 | 3 |
|---|---|---|---|
| B1 | .0063 | .0037 | .0012 |
| B2 | .0080 | .0027 | .0028 |
| B3 | .0082 | .0028 | .0027 |

Table 8. 3 Subject Experiment

Notice that even using 10% of the total number of mutants, the strength of the test data is within 1% of the adequate set. This experiment was repeated using the programs cited in another study (see Chapter 6). In each case it was determined that the test data remained within 1% of the adequate test data.

These experiments suggest strongly that a cost effective approach to generating adequate test data is to generate only a small percentage of the total number of mutants and develop test data which is adequate relative to this set of mutants.

## Efficiency and Redundancy in Operators

The results quoted above dealing with random sampling of the mutants might measure still another effect: redundancy among the operators. That is, it may be possible to derive strong test data from a random subset of the mutants simply because so many mutations deal with the same error or type of error. Thererfore, it is natural to look for efficiency in the mutation process by eliminating those mutants from consideration which do not add significantly to the strength of the test data generated.

For an operator to be useful it must force the tester in some way to produce stronger test data than could have been produced without it. If all of the mutation produced by a given operator are eliminated by virtually any test data that executes the affected line, then it is natural to assume that the operator does not significantly improve on the statement coverage operators.

Let us fix a mutation operator and define the following parameters. $N_t$ is the total number of mutants generated by that operator, $N_u$ is the number of mutants that are eliminated on the first execution by a given data set, and $N_e$ is defined to be the number of equivalent mutants.

A measure of efficiency for such an operator is given by

$$(N_t-(N_u+N_e))/N_t.$$

Notice that $N_t$ and $N_e$ depend only on the program being considered and the mutation operator. $N_u$ depends on the choice of test data being supplied. The redundancy of a mutation operator is then given by:

$$(N_u+N_e)/N_t.$$

A procedure for collecting operator efficiency data is the following. First, select several programs representative of the space of programs in the intended application. Second, generate test data that is just strong enough to execute all statements. Third, generate test data to obtain a mutation score of 1. The point of the second step is to intentionally produce weak tests, which force statement coverage but do as little other testing as possible.

After such measurements have been made on several programs and for multiple independent test data generations for each program, a set of efficiency measurements for each operator will be obtained. If an operator consistently has a high redundancy, then the deletion of the operator from the system appears justified. An operator possessing high efficiency on all programs and all test sets evidently forces the tester toward stronger test data and should be retained.

The approach outline above has two limitations. First, it does not consider <u>interactions</u> between operators. That is, operators may have the same high efficiencies, but each actually has the same effect. In this case, one or the other may be necessary, but certainly not both. The efficiency measurements will not give an indication of this condition since they provide only the interaction of the TRAP operator with all of the others. Therefore, the experiment can be widened to indicate operator redundancy with any subset of the operators by replacing step 2 of the data gathering procedures with the following: generate test data just strong enough to eliminate all of the nonequivalent mutants generated by the given subset of error operators. Of course, the definition of $N_u$ needs to be accordingly modified.

Ideally, we would like to measure the efficiency of operators relative to all possible subsets in order to find the minimal set of operators which delivers adequate tests. Since this is not feasible, a less demanding strategy is required. For example, it is possible to choose the most efficient operator relative to TRAP, then choose the most efficient relative to TRAP and the first operator, and so on. The process terminates when there is no remaining operator whose efficiency relative to the set chosen is above a given threshold.

Obviously, this approach applies only to a given class of program from which the sampling takes place. Changing the language or even the programming discipline might effect operator efficiency. However, if the sample population is representative it is always possible to "tune" the set of operators for that population by using only operators which derive useful testing information.

The results of a single data generation experiment for the Cobol program A1-A6 are given in the following table. An asterisk indicates that no mutants of that type were generated for the program.

| Operator | Program | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| Decimal | * | 0.96 | 0.30 | 0.21 | 0.33 | 0.18 |
| Occurs | * | * | * | 0.00 | * | * |
| Insert | 0.00 | 0.00 | 0.01 | 0.00 | 0.00 | 0.00 |
| Fill.Siz | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Item Rev | 0.05 | 0.04 | 0.07 | 0.00 | 0.00 | 0.01 |
| Delete | 0.00 | 0.34 | 0.00 | 0.01 | 0.04 | 0.03 |
| Go-Perf. | * | * | * | 0.00 | * | 0.00 |
| Perf.-Go | 0.00 | 0.00 | 0.00 | 0.00 | 0.08 | 0.00 |
| IF Rev. | 0.00 | 0.67 | 0.00 | 0.06 | 0.00 | 0.00 |
| Stop | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Thru | 0.00 | * | * | 0.06 | * | 0.00 |
| Arith | * | 0.75 | * | 0.04 | 0.05 | * |
| Compute | * | 0.50 | 0.25 | * | 0.00 | 0.00 |
| Parenth. | * | * | 0.00 | * | 0.00 | 0.00 |
| Round | * | 0.44 | 0.20 | 0.00 | 0.11 | 0.17 |
| Move Rev | 0.00 | 0.00 | 0.00 | 0.00 | 0.04 | 0.01 |
| Logic | 0.07 | 0.51 | 0.00 | 0.13 | 0.24 | 0.05 |
| SFS | 0.01 | 0.34 | 0.03 | 0.01 | 0.04 | 0.02 |
| CFC | 0.00 | 0.25 | 0.00 | 0.01 | 0.10 | 0.04 |
| CFS | 0.00 | 0.36 | 0.03 | 0.01 | 0.05 | 0.04 |
| SFC | * | 0.18 | 0.00 | 0.03 | 0.09 | 0.04 |
| C Adjust | 0.00 | 0.50 | 0.14 | 0.06 | 0.22 | 0.03 |
| Files | 0.00 | * | * | * | * | 0.00 |

Table 8. Operator Efficiency Data

There is obviously a wide variation in efficiencies between the programs. This a partly due to the indirect test data selection procedures and partly due to the inherent differences in the programs.

The first five operators are of special interest . These are Cobol data mutations that force the system into interpretive execution using a run-time symbol table. If these mutants can somehow be eliminated; then a more efficient compiled execution of mutant is feasible. The first operator moves the implied decimal point in a numeric item. It is useful primarily in that it forces the tester to provide nonzero values for that variable. The same effect can be achieved by an operator which resembles ZPUSH. The second operator alters the OCCURS count in a table description. Since the sample programs make little use of tables, nothing can be inferred from the data for this operator. Inserting an extra filler in a record is of little use, as is altering the size of a filler. Reversing two adjacent elementary items within a record is sometimes a useful operation, but the same effect can most likely be achieved by substituting one field for another in the procedure division.

In the procedure division, changing a GOTO to a PERFORM usually provides no testing power. Perhaps most of the testing effort in trying various path alternatives is already achieved by simple statement coverage. Inserting a STOP statement is not helpful because in most program files, files will be left open which is an error. STOP insertion thus play essentially the same role as TRAP. THRU clause alteration, reparenthesization of arithmetic expressions and the reversal of the direction of a binary MOVE and changing an I/O reference from one file to another are also rarely useful in this study. It may be that these mutations are too drastic. Errors this large may be detected by almost any test case that exercises all program statements. The errors sought after simple statement coverage are rather more subtle ones. The major errors have already been ruled out.

A non-redundant set of Cobol operators then might be the following: statement deletion, IF reversal, and the substitution operators for arithmetic operators, scalar for constants, constants for scalars, constants for constants, scalars for constants, and constant adjustment.

## Bibilographic Notes

An overview of practical experiences with mutation analyzers which support the analytic and experimental bounds discussed in this chapter can be found in the papers [Acree, 1979],[Acree, 1980], and [Budd, 1980]. The data relating to the number of mutants generated as a function of program size was developed by Acree, Budd, DeMillo, Lipton and Sayward and is reported in [Acree, 1979]. The data relating complexity with the number of mutants appears in Budd's thesis [Budd, 1980].

Experimental results on mutant stability and the effectiveness of sampling have been treated by Budd and Acree in [Acree, 1980] and [Budd, 1980] and are also reported in [Acree, 1979].

The notion of operator efficiency was developed in Acree's thesis [Acree, 1980].

# Chapter 6

## Further Experimental Studies

In experimental studies of program testing, the problems of interest are:

1. What is the cost of performing the test?
2. What is gained from performing the test?

In general, quantitative answers to these questions are the most desirable, but that seems to be beyond the state-of-the-art. A less precise but still valuable solution is to discover how testing costs relate to the performance of the test. In practice, this cost-benefit ratio is the one that will be of most use in determining which testing technique to apply.

The cost of program mutation is ultimately constrained by the number of mutants which must be executed. As described in previous chapters, the set of mutants $\mu$ of a program is defined by a set of mutant operators that result in a set $\mu$ whose size is bounded roughly by the product of the number of data references and the number of distinct data references. As discussed in Chapter 5, it is generally not necessary to execute all mutants in $\mu$, since random sampling yields test data whose mutation score is only slightly inferior to an adequate test set.

However, one should question the effectiveness of applying program mutation with only simple mutants since other more complicated (but reasonable) alternatives are apparently overlooked. This is an apparent violation of the Competent Programmer Assumption. The coupling effect indirectly addresses the more complicated mutants of P: test data that causes all simple mutants of P to fail is so sensitive that it implicitly causes all complex combinations of them to fail. In Chapter 3, we examined two situations in which error coupling guarantees that test data adequate for a simple set of mutants is also adequate for mutants which satisfy the Competent Programmer Assumption. In this chapter we will examine some experimental evidence for the address the observable properties of error coupling.

## Beat the System Experiments

Evidence against error coupling is any event in which incorrect program are successfully tested against an adequate test set. Since such examples can always be "cooked-up" for any test technique, a problem of more practical importance may be what kind of errors are always detected and what kind of errors are overlooked.

At present these questions can only be studied empirically because of the lack of any widely accepted formal models of programming errors.

One sort of experiment is a many-subject experiment. The experiment has N subjects with varying levels of programming and testing skill and M programs that have zero or more errors known only by the experimenter, and each subject reports on the errors detected in trying to pass the mutant test.

Another useful experimental technique is a single-subject experiment. We call such an experiment a <u>beat the system</u> experiment. The single subject is someone having a very high level of programming expertise and much familiarity with the concepts of program mutation. The M programs have one or more errors, and the subject has <u>complete knowledge</u> of what the errors are. The subject tries to beat the mutation system — to pass the mutation test with an incorrect program by developing test data on which the program is correct but on which all mutants of the program fail. If there are error types for which the highly skilled subject cannot beat the system, then these error type will probably be detected by any user of the system. On the other hand, if there are error types for which the subject can consistently beat the system, then the given set of mutant operators has a certain weakness in detecting these errors.

A beat the system experiment is an attempt at a worst-case analysis. We attempt to find out how the system will perform under the worst system circumstances. Beat the system experiments are extensions of experimental reliability studies. A testing technique is said to be <u>reliable</u> for for an error type if the use of the testing technique is <u>guaranteed</u> to reveal the presence of the errors of that type. Reliable studies are aimed at comparing two or more competing methodologies and deriving statistical information of the form "On the following examples of programs, method A discovered X% of the errors and method B discovered Y%." In the beat the system experiments we are more concerned with the type of errors missed.

For example, several of the programs studied in early experiments revealed that a significant number of errors in Fortran are caused by programmers' treating the DO statement as if it were an Algol FOR statement. These errors are detected by introducing a mutant that changes a DO statement into a FOR statement, bringing this fact to the programmer's attention and forcing him to derive data that indicates he had knowledge of this potential pitfall.

We will describe two sets of experiments. The first set is a beat the system experiment using the Fortran programs B1-B11. These programs are described in Appendix B. Appendix B also contains descriptions of the errors in these programs. The second set of experiments adapts earlier reliability studies in a comparative analysis of program mutation and a number of other testing techniques.

It is difficult to construct a classification scheme for error types that is neither so specific that each error forms its own type nor so general that important patterns cannot be detected (cf. Chapter 2). If the classification is based on logical mistakes, then it is often hard to relate errors to mistakes in the code. On the other hand, it seems difficult to base a scheme just on mistakes in the code, since often a single logical mistake will be responsible for changes in several locations in the program. Following the classification scheme in Chapter 2, we group errors into the following categories:

**Missing path errors**: These are errors where a whole sequence of computations that should be performed in special circumstances is omitted.

**Incorrect predicate errors**: These are errors that arise when all important paths are contained in the program, but a predicate that determined which path to follow is incorrect.

**Incorrect computation statement**: These are errors that arise from a computation statement that is incorrect in some respect.

**Missing computation statement**: These are errors that arise from the omission of one or more computational steps.

**Missing clause in predicate**: This is a special case of an incorrect predicate error, but, since it is hard to detect, we give it special treatment.

The 25 errors in the program B1-B11 range from simple to subtle errors. Because of the worst-case nature of the experiment, the fact that 5 errors are not discovered does not mean that these errors would always remain undiscovered if mutation analysis was used in a normal debugging situation. Table 1 gives the number of errors detected by error type. Of these 25 errors, only 8 would be caught using branch analysis.

| Error Type | Number | Caught |
|---|---|---|
| Missing path error | 6 | 5 |
| Incorrect predicate error | 3 | 2 |
| Incorrect computation statement | 12 | 11 |
| Missing computation statement | 3 | 2 |
| Missing clause in predicate | 1 | 0 |

Table 1: Number of errors detected by error type

In three of these categories, the errors are caused by the lack of certain constructs in the program. Since the testing method is asked to guess at something that is not in the program, we should really be surprised that it does as well as indicated. Nonetheless, missing path errors and missing clauses in predicates are probably the most difficult errors for any testing method to discover.

The failure of the mutation in detecting these 5 errors is probably not an indication of a weakness in the method, rather, it reflects on our choice of mutant operators. It is quite possible that with another set of mutant operators many of these errors would be caught.

The second experiment is derived from an earlier reliability study by Howden and uses two sources of data. The first is the book Elements of Programming Style by B. Kernighan and P. Plauger. In a chapter entitled "Common Blunders" Kerighan and Plauger offer twelve program fragments, each containing errors inserted to illustrate common programming mistakes. In a beat the system experiment, these twelve program fragments were subjected to symbolic evaluation, path analysis (each loop executed at least twice), a combination of symbolic evaluation and path analysis, and program mutation. Once path domains are identified, the experimenter uses a random choice of test data for the domains. Therefore, it is possible that more sensitive input partition tests will yield slightly different results.

The following table summarizes the results of this experiment

| Test Method | Error Caught | Total Error |
|---|---|---|
| Symbolic Evaluation | 13 | 22 |
| Path Testing | 9 | 22 |
| Combine Methods | 16 | 22 |
| Program Mutation | 20 | 22 |

Table 2. First Reliability Study

The 20 errors detected by program mutation are detected in six ways. The interpreter of an automated mutation analyzer was responsible for detecting 8 errors, 5 were detected by spoiling coincidental correctness expressions (cf. Chapter 10), 2 were caught by finding a correct mutant of the incorrect program, 2 are caught by ABS insertion, two are detected by predicate testing (see Chapter 4) and 1 error was detected by an explicit branch analysis mutant. The two errors not detected consisted of a two statement interchange in a routine for computing the sine function and an error involving an equality test between reals. The following table describes the errors and the mutants which detect them.

| Error | Method of Detection |
|---|---|
| variable SUM uninitialized | interpreter |
| DABS operator needed | explicit mutant |
| -1**(I/2) used instead of (-1)**(I/2) | I/2 ==> I/1 or I/2 with no effect |
| interchange of statements | not detected |
| variable E uninitialized | interpreter |
| type mismatch | interpreter |
| variable C not reset | to eliminate branch analysis mutants, SC+CI must be less than or equal to TC |
| error when CI = 0 | caught by ZPUSH mutant |
| expression should be NUM(1) | interpreter |
| override of DATA statement initialization | interpreter |
| failure on 46 transactions | $>$ ==> $\geq$ |
| $\geq$ should be $>$ | $\geq$ ==> $>$ |
| undefined variable | 1==>2 on lower DO loop limit |
| error if B+C $<$ .01 | twiddle B+C by .01 |
| loop exits incorrectly | increase iterations by 1 |
| uninitialized variable | interpreter |
| one entry tables cause error | (LOW+HIGH)/2 ==> LOW+HIGH-2 |
| failure to match A(1) | (LOW+HIGH)/2 ==> LOW+HIGH-2 |
| J=MARKS(I)-1/10 should be J=(MARKS(I)-1)/10 | I/10 ==> 0/10 |
| missing parthentheses around expression AN-1.0 | ZPUSH (SUMSQ-(SUMSQ**2/AN)) |
| 10*.1 = 1 | caught by all data |
| equality test on reals | not detected |

Table 3. Mutants Detecting Errors

Error 19 is one of the errors not detected by either path analysis or symbolic evaluation, although a symbolic evaluator with a special two dimensional output could have caught the error. In Fortran, the expression 1/10 evaluates to 0. Therefore, the mutant which replaces 1/10 with 0/10 catches the error. Neither path analysis nor symbolic evaluation detect error 2, which is an explicit mutant of a correct program.

A second experiment uses the programs B1 - B4 in a comparison of the error detection capabilities of path analysis, branch analysis, functional testing, special values testing, anomaly testing, and black-box analysis. The path analysis discipline for this experiment requires each loop to be executed at least once. Special values testing is a collection of heuristics (e.g., force every expression to 0).

Table 4 presents the results of this experiment.

| Test Method | Error Caught | Total Errors |
|---|---|---|
| Path Analysis | 4 | 5 |
| Branch Analysis | 0 | 5 |
| Functional Testing | 3 | 5 |
| Static Analysis | 0 | 5 |
| Black Box Testing | 3 | 5 |
| Program Mutation | 4 | 5 |

Table 4. Second Reliability Study

The error which was not detected by program mutation is a missing path error (see Appendix B). Apparently these errors are the most difficult for dynamic testing techniques. On the other hand test techniques which work from functional descriptions or specifications of program behavior seem to do quite well at detecting these errors.

## Experiments on the Coupling effect

We begin with an example of the experimental evidence for the existence of error coupling.
The subject program is Hoare's
FIND program (see Appendix B, Program B10). FIND was used in the following experiment.

1. A test data set of 49 cases was derived and shown to be adequate.
2. The test data set from 1 was heuristically reduced to a set of 7 test cases which also turned out to be adequate.
3. Random simple k-order mutants were selected ($k > 1$).
4. The higher order mutants of step 3 were executed on the reduced test data set.

It would be evidence against the coupling effect if it was possible to randomly generate very many higher order non-equivalent mutants on which the reduced test data set behaved in a manner indistinguishable from FIND. Notice that Step 2 biases the experiment against the coupling effect since it removes the man-machine orientation of mutation analysis. We concentrated first on the case $k=2$, with the following results:

| Property | Number of Mutants |
|---|---|
| 2-order mutants | 21,100 |
| indistinguishable from FIND | 19 |
| equivalent to FIND | 19 |

Table 6. 2-order Mutants

However, a limited analysis of higher order mutants produced the following results:

| Property | Mutants |
|---|---|
| Number of k-order mutants (k>2) | 1,500 |
| Number indistinguishable from FIND | 0. |

Table 7. Higher Order Mutants

The following argument shows a defect in this experiment. Just as the competent programmer assumption states that programs are not written at random, the coupling effect is implied by the fact that program statements are not composed at random; indeed, there is considerable flow and sharing of information between statements of a program, so that a change to one portion of a program is likely to have observable, albeit subtle, effects on its global context. Now for the problem with this experiment: the k-order mutants are chosen randomly and by independent drawings of 1-order mutants. Therefore, the resulting higher-order mutant is very unstable and subject to quick failure. The experiment should also be conducted when the higher-order mutants contain subtley related errors. To this end, the experiment was repeated using the following replacement for step 3:

3': Randomly generate correlated k-order mutants of the program.

In Step 3', "correlated" means that each of the k applications of 1-order mutant operators will be related in some way to all of the preceding applications, all affecting the same line, for example. As before, if a program is successfully subjected to mutation analysis on a test data set, then the coupling effect asserts that the correlated k-order mutants are also likely to fail on the test data.

To broaden the experiment we use, in addition to FIND, the programs (B12) STKSIM which maintains a stack and performs the operations clear, push, pop, and top, and TRIANG (B9) which classifies integers as either not representing the lengths of sides of any triangle or as representing the sides of scalar, isosceles or equilateral triangles.

Table 8 contains a summary of the results of the experiment. The data suggests strongly that there is a meaningful sense in which errors are coupled by an appropriate choice of error operators.

| PROGRAM NAME | k = 2 | | k = 3 | | k = 4 | |
|---|---|---|---|---|---|---|
| | NUMBER GENERATED | NUMBER ALIVE | NUMBER GENERATED | NUMBER ALIVE | NUMBER GENERATED | NUMBER ALIVE |
| FIND | 3000 | 2 | 3000 | 0 | 3000 | 0 |
| STKSIM | 3000 | 3 | 3000 | 0 | 3000 | 0 |
| TRIANG | 3000 | 1 | 3000 | 1 | 3000 | 0 |

Table 8. Correlated k-order Mutants

The results are for the most part self explanatory. Except for the correlated three-order mutant of TRIANG, all of the correlated k-order mutants described in the table are equivalent to their subject programs. The remaining live TRIANG mutant would have been eliminated with a more sophisticated error operator for detecting loop boundaries.

Essentially the same study was repeated using A1-A6. The basic format of the experiment remained the same: develop adequate test data, randomly generate a large number of complex mutants, execute the selected mutants on the test data, keeping track of those not eliminated, and remove equivalent mutants from the list of uncoupled complex mutants.

In all cases the strategy in randomly selecting complex mutants was to use uniform sampling with replacement from the given space of complex mutants. The parameters of each experiment are the program being tested, the tester, the types of complex mutants considered and the sample size. It is possible that the effects of the human tester are relevant. The repetition of this experiment by other investigators should determine the variation in the strength of error coupling due to test data generation.

As before, we concentrate on second order mutants, both correlated and uncorrelated. The statistic that is developed is a confidence interval on the fraction of second order mutants that are uncoupled. Since error coupling is not expected to be total in practice, this gives us an estimate of the probability that a second order mutant escapes detection by mutation analysis. If we find any uncoupled mutants, we obtain a two-sided confidence interval and if we find none we still obtain a one-sided — upper bound -- confidence interval.

For the experiments with uncorrelated pairs of mutants, a sample size of 50,000 meaningful second order mutants was used for each of the six programs. Table 9 summarizes the results.

| Program | Pairs Survive 1st Order Test Data | Not Equiv. | 95% Confidence Interval on (z 10,000)* |
|---|---|---|---|
| A1 | 26 | 0 | 0.0 -- 7.4 |
| A2 | 12 | 0 | 0.0 -- 7.4 |
| A3 | 22 | 5 | 3.2 -- 23.3 |
| A4 | 10 | 2 | 0.5 -- 14.4 |
| A5 | 45 | 0 | 0.0 -- 7.4 |
| A6 | 13 | 0 | 0.0 -- 7.4 |

Table 9. 50,000 Uncorrelated Mutants

Test data generated to kill first order mutants proved to be sufficient to kill at least 99.976% of all second order mutants in all cases considered, and 99.992% in most cases. Significantly, program size does not seem to be an important factor in the strength of error coupling. If these results hold over a broad range of programs, the addition of second order mutants can be expected to give almost no additional power not already present in simple mutants, and certainly not enough to justify their cost.

The experiments on second order mutants used 10,000 mutants for each program. The format of the experiments is otherwise identical to the ones above. The results of these experiments are summarized in Table 10.

| Program | Pairs Survive 1st Order Test Data | Not Equiv. | 95% Confidence Interval on (x 10,000)* |
|---|---|---|---|
| A1 | 0 | 0 | 0.0 -- 36.9 |
| A2 | 3 | 1 | 0.3 -- 55.7 |
| A3 | 60 | 19 | 114.4 --296.6 |
| A4 | 3 | 3 | 6.1 -- 87.6 |
| A5 | 1 | 0 | 0.0 -- 36.9 |
| A6 | 1 | 0 | 0.0 -- 36.9 |

Table 10. 10,000 Correlated Mutants

The same six programs were subjected to a final series of experiments to look for uncoupled mutants of orders 2 through 5. 20,000 complex substitution mutants were generated for each program and each order. Intuition suggests that it is not necessaary to carry out such experiments for extremely large values of k: the more errors introduced into a program, the more the Competent Programmer Assumption is violated. On the other hand, the behavior of extremely high order mutants is not well understood, and it seems prudent to examine some data on multiple mutations, if only to insure that there are no unexpected processes at work.

For this experiment, 20,000 complex substituion mutants of order k ($2 \leq k \leq 5$) were generated for each of the six Cobol programs. All mutants examined were uncorrelated. The mutants were randomly selected and then examined to insure that all mutations applied to distinct data references. The folowing table shows the number of mutants that passed the first order test data for each program, and the number that were not equivalant — these are uncoupled mutants.

| | | Program A1 | A2 | A3 | A4 | A5 | A6 |
|---|---|---|---|---|---|---|---|
| 2nd Order Mutants | Number that Pass Test | 1 | 2 | 5 | 0 | 9 | 5 |
| | Uncoupled Errors (Noneqiv.) | 0 | 0 | 1 | 0 | 0 | 0 |
| 3rd Order Mutants | Number that Pass Test | 0 | 0 | 0 | 0 | 0 | 0 |
| | Uncoupled Errors (Noneqiv.) | 0 | 0 | 0 | 0 | 0 | 0 |
| 4th Order Mutants | Number that Pass Test | 0 | 0 | 0 | 0 | 0 | 0 |
| | Uncoupled Errors (Noneqiv.) | 0 | 0 | 0 | 0 | 0 | 0 |
| 5th Order Mutants | Number that Pass Test | 0 | 0 | 0 | 0 | 0 | 0 |
| | Uncoupled Errors (Noneqiv.) | 0 | 0 | 0 | 0 | 0 | 0 |

Table 11. Higher Order Mutants

**Uncoupled Errors:**

The uncoupled errors discovered in the last three series of experiments described above involved alterations to predicates in conditional expressions. They can be classified as follows.

**Type I Errors:** Changing both operands in a comparison

IF(a operation b) ==> IF(a' operation b')

**Type II Errors**: Changing an operand and operation in a comparison

IF(a operation b) ==> IF(a' new-operation b)

**Type III Errors**: Changes to non-interacting comparisons

IF($P_1$(a) $\wedge$ $P_2$(b) $\wedge$ ...) ==> IF(NOT $P_1$(a) $\wedge$ $P_2$(b) $\vee$ ...)

If an uncoupled error is thought of as a potential error in the program, then these three types of uncoupled errors represent a form of coincidental correctness (see Chapter 10): taking the right path for the wrong reason. A plausible reason that these are the only known types of uncoupled errors is that mutation analysis does not explicitly test higher level path coverage. Indeed the problem of testing higher level path coverage is so complex (due simply to the number of paths) that it is probably out of reach of _any_ systematic testing technique.

## Coupling and Complexity Measures

There are frequent references in the literature to a possible relationship between program reliability and structural characteristics of the program. If such a relationship exists, then it is possible that there is a similar relationship between those structural characteristics and error coupling. One such characteristic is structural complexity, measured, for instance, by the number of program branches).

Consider the following simple test strategy, often called DD path coverage. The goal is to develop test data that forces the program down every path from decision point to decision point. This strategy may require test data which drives the program down a particularly complex path to discover an error. For example, consider the following program, which sorts the triple (A,B,C).

```
L1: if A < B then goto L2;
    T:=A; A:=B;B:=C;
L2: if B < C then gotto L3;
    T:=A; A:=C;C:=T;
L3: if B < C then goto L4;
    T:=B; B:=C;C:=T;
L4: stop
```

The program is incorrect. The condition at L2 should be A<C. The input (1,2,3) and (3,2,1) both give correct results and force the execution of all decision to decision branches. (1,2,3) takes the TRUE branches at L1-L3 while (3,2,1) takes the FALSE branches. The error is not uncovered in this way: what is needed is a test case that forces execution of a complex path corresponding to differing outcomes at L1 and L2 . Thus simply covering all branches leaves some errors undetected. It is possible that mutation contains the same weakness, since mutations tend to be localized in the program (note, however, that mutation analysis contains DD path coverage as a special case, so it can be no weaker; cf. Chapter 2). The number of test cases required for exhaustive testing of all possible conditions in this program is $2^3 = 8$.

To test the relationship between the number of branches and error coupling, we hypothesize that the more branches a program has, the harder it is to develop adequate test data. In more concrete terms: the proportion of uncoupled errors rises with the structural

| Program | Number of Branches | Number of Records | Number of Mutants | Number that Pass | Number Uncoupled |
|---|---|---|---|---|---|
| C-1 | 0 | 1 | 474 | 329 | 0 |
| C-2 | 1 | 3 | 480 | 153 | 1 |
| C-3 | 3 | 7 | 492 | 84 | 1 |
| C-4 | 5 | 12 | 504 | 50 | 3 |
| C-5 | 7 | 15 | 516 | 18 | 9 |

Table 12. Complexity Metric Data

Eleven of the surviving uncoupled mutants are of type I. The other three are of type II. The relatively large number of equivalent mutants in these programs is due to the padding that was complexity of the program. An experiment to test this hypothesis would match program for length and number of mutants and would allow the branching count to vary, measuring the coupling coefficient, defined in Chapter 2.

If the confidence intervals on the estimates of the coefficients overlap, then no relationship may be inferred. If there is no overlap, then there is a statistical relationship. If, in addition, there is a causal mechanism responsible for the statistical relationship, an argument could be made for simplicity in program structure for program to be tested by program mutation.

For this experiment, a sequence of small programs was written, all using the same data items and data references, but with an increasing number of branches. The experiments examined 50,000 pairs of mutants for each program. The following table shows the number of branches, test cases , mutants, pairs passing the test data and uncoupled mutants for each program used to insert extra branches without greatly affecting the number of mutants generated. The 95% confident interval on z(100,000) plotted against the number of branches is shown in Figure 1.

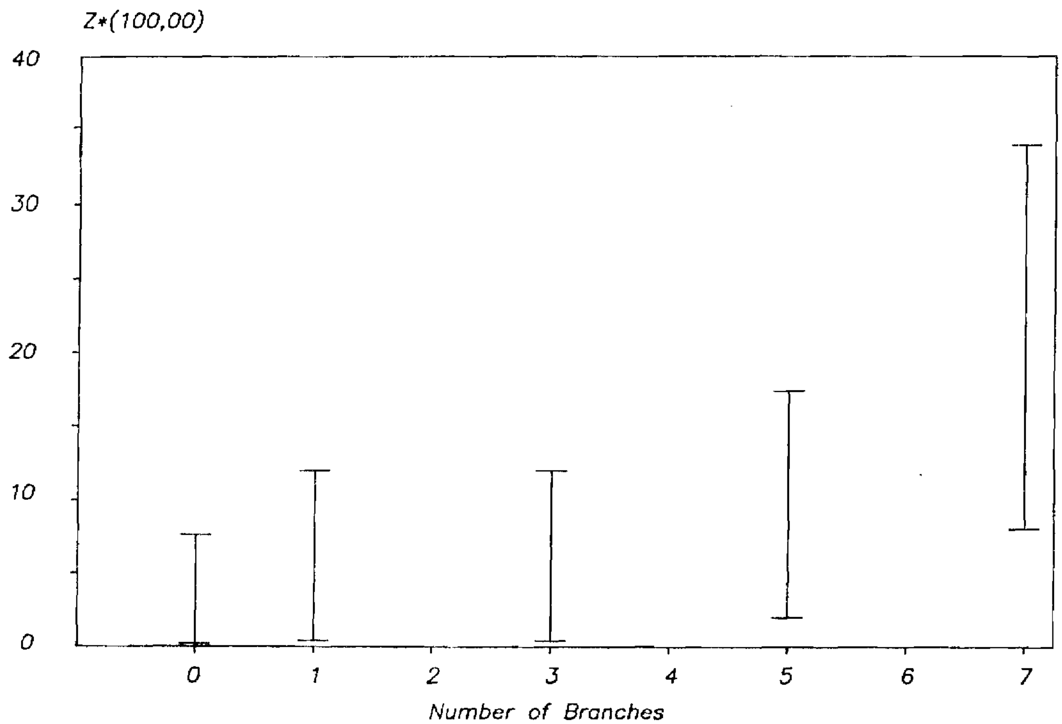



Figure 1.
95% Confidence Intervals

It is apparent that in this set of programs, the effect of adding complexity is very slight. It can be accounted for by the type of uncoupled mutants seen in the experiments described above. If this relationship holds in practice, then the branching complexity of programs has little impact on the difficulty of mutation analysis.

## Bibliographic Notes

The beat the system experiments were designed by Budd and Sayward. The data reported here is taken from Budd's thesis [Budd, 1980] and a paper by Budd, DeMillo, Lipton and Sayward [Budd, 1980b]. The experiments on the coupling effect were designed by Acree [Acree, 1980] and DeMillo, Lipton and Sayward [DeMillo, 1978a]. The data also appeared in [Acree, 1979]. Experiments on program complexity were carried out by Acree [Acree, 1980].

# Chapter 7

## Mutant Equivalence

Experience indicates that in production programs, the number of equivalent mutants can vary between 2% and 5% of the total mutant count. In more finely tuned programs, however, it is common for source statements to appear in a particular form solely for efficiency reasons. In these program such statements can be altered without affecting the output behavior. A typical example of this behavior is beginning a loop at 2 instead of 1 or 0, so that a mutation which changes "2" to "1", for example, causes an extra iteration but does not alter the outcome of the looping operation. In tuned programs, the equivalent mutants can comprise as much as 10% of the total.

Equivalent mutants are not distributed with respect to their operators in the same proportion as other mutants. In fact, a samll number of mutant types account for the preponderance of equivalent mutants. The following table provides some data on the distribution of equivalent mutants for typical Fortran programs.

| Mutant Type | % Equiv. | % of all |
|---|---|---|
| Absolute Value Insertion | 75 | 4.0 |
| GOTO Replacement | 12 | 0.7 |
| Relational Operator Replacement | 5.5 | 0.5 |
| All Other Mutant Types | 5.5 | 0.5 |

Table 1. Distribution of Equivalent Mutants by Type

It has become increasingly clear that determining mutant equivalence ranges from very difficult to very easy. It is helpful to classify the types of equivalence which must be judged. At the first level are mutants which are detectable as equivalent by noting that (1) if a parameter has a variable upper bound, the value of the upper bound must be positive, and (2) the values on loop variable limits determine the range of values of the loop variable for the extent of the loop. At the second level are mutants which can be judged equivalent by examining.

It is easy to show that equivalent mutant detection is an undecidable problem Assume a fixed programming language which is expressive enough to allow the programming of all recursive functions, and let P1 and P2 be arbitrary procedures written in the language. Since "goto" mutations are meaningful and likely mutations, consider the following program to which **goto** replacement has been applied.

```
    goto L;                        go to M;
 L:P1;halt;          ==>        L:P1;halt;
 M:P2;halt;                     M:P2;halt;
```

Clearly, these two programs are equivalent (that is, they either halt together and deliver the same output or they diverge together) if and only if P1 and P2 are equivalent, and that is undecidable for the language described above.

In spite of this, most equivalent mutants which arise in practice are stylized and rather easy to judge equivalent. This is perhaps due to the Competent Programmer Assumption: the subject program and an allegedly equivalent mutant are not chosen randomly — in fact, they are chosen by a very careful sieving of all possible programs and the structure of this relationship should be something that one can exploit in determining mutant equivalence.

## Human Evaluation of Equivalence

It would be desirable to measure in an experimental setting the accuracy of human testers in judging mutant equivalence. This section describes an experiment conducted using the programs A-3, A-4, A-5, and A-6 . For each program, a sequence of test cases was used to eliminate mutants, but testing was stopped when the number of mutants remaining was approximately twice the number of remaining mutants. This process eliminated most of the obviously inequivalent mutants. From the remaining mutants, for each program, a subset of fifty mutants was randomly selected. Two subjects were used in this experiment.

Both subjects had been involved in the development of mutation nalysis systems, and both were competent programmers. Neither subject had been exposed to the programs used in the experiment. Each subject was given the list of mutants and the source listing for each of the programs and was instructed to mark each mutant equivalent or not equivalent. There were no other intructions or restrictions placed on the subjects.

There are two kinds of errors that can be made in judging equivalence. The first type of error is the marking of a non-equivalent mutant as equivalent. The second type of error mistakes equivalent mutants as non-equivalent. Errors of the second type are not very serious, since in the process of mutation analysis, the mutant remains in the system and can be reconsidered at any later time. However, when a type 1 error occurs, a mutant which can be valuable in detecting errors is prematurely removed from the system. Premature removal of mutants increases the likelihood that an erroneous program will be accepted as correct by the tester.

The results of human evaluation of the four programs is shown in the following table.

| Program | No. Equiv | No. Not | Subject 1 | | | Subject 2 | | |
|---|---|---|---|---|---|---|---|---|
| | | | Correct | Type 1 | Type 2 | Correct | Type 1 | Type 2 |
| 3 | 20 | 30 | 44 | 0 | 6 | 42 | 2 | 6 |
| 4 | 21 | 29 | 36 | 2 | 12 | 33 | 6 | 11 |
| 5 | 20 | 30 | 46 | 0 | 4 | 40 | 5 | 5 |
| 6 | 13 | 37 | 33 | 16 | 16 | 45 | 1 | 4 |

Table 2. Human Evaluation of Equivlance

The tables show the number of equivalent and non-equivalent mutants in the mutant sample present late in the testing process, and the number of correct identifications of errors. More significantly the table documents the number of errors of each type in judging mutant equivalence.

Subject 1 was more variable in accuracy that Subject 2, but overall their results were similar. Subject 1 identified 79.5% of the mutant correctly. Subject 2 was correct on 80% of the mutants. In measuring type 1 errors the best computation is probably the total type 1 errors as a percentage of the total number of non-equivalent mutants, since these represent the potential type 1 errors. Subject 1 made type 1 errors on 14.3% of the non-equivalent mutants, while Subject 2 made type 1 errors on 11.1%. On the other hand, Subject 1 made type 2 errors on 31.5% of the equivalent mutants, and Subject 2 made type 2 errors on 35.1%.

The number of type 1 errors may be high enough to significantly reduce confidence in the abilities of human evaluators if it is an accurate reflection of the frequency of such errors in practice. It should be remembered, however, that the subjects were required to mark each mutant as equivalent or not with only the evidence at hand (the source listing), while a tester in practice may postpone the decision pending further testing and thought. In addition, the subjects worked in isolation and thus were denied both helpful consultation and the motivation of accountability for potential errors. These are important factors in actual testing situations. High error rates for type 2 errors indicate that the subjects were being conservative in their judgements, marking mutants as non-equivalent when in doubt.

This observation leads us to consider automated techniques for judging mutant equivalence. An automated technique will have the desirable properties of the human evaluators. Namely, an automated technique will make type 2 errors. On the other hand, an automated equivalence tester never makes type 1 errors.

**Automated Equivalence Checking**

Before we proceed it may be instructive to examine a few instances of equivalent mutants which show this structure. In the analysis of the FMS.1 scanner (see Section 2), a relatively large number of mutants resulting from the transformation

```
X ==> RETURN
```

appear as live mutants on even very good test data. On closer examination, however, most of these reveal that

```
X = GO TO 90,
```

where statement labelled 90 is itself a RETURN. The programmer's style is to always jump to a common RETURN statement, allowing an easy "proof" of equivalence.

For another example, let us return to the NXTLIV routine described in Chapter 9. A principal source of equivalent mutants in that example was the troublesome test for a word of zeroes. Its only purpose is to save the effort of looking through the words bit by bit. If the condition in the test is replaced by any identically true expression,

```
IF(L.NE.0)GOTO 23 ==> IF(12.NE.0)GO TO 23
```

the program runs a bit longer but is otherwise identical. Similarly the mutation

```
IF(MUTNO.GT.MCT)GOTO 40 ==> IF(MUTNO.GE.MCT)GOTO 40
```

changes the performance of the program only, but this time it improves it!

These last two examples are not accidental. Mutations of a program are similar to simple transformations that are made in code optimization; it is not surprising that some of them should turn out to be optimizing or de-optimizing transformations. Conversely, correctness preserving optimizing transformations should be applicable to detecting equivalent mutants. If this is a useful heuristic then the task of identifying equivalent mutants can be reduced to detecting those which are equivalent for an interesting reason.

Almost all of the techniques used in optimizing compiled code can be applied in some way to decide whether a mutant is equivalent to the subject program. Some optimizing transformations are widely applicable while others are limited in scope. We will give a sampling of the useful transformations.

**Constant Propagation:** Constant propagation involves replacing constants to eliminate run-time evaluation. A typical optimizing transformation would replace statement 3 as shown below

```
1    A=1             1    A=1
2    B=2     ==>     2    B=2
3    C=A+B           3    C=3
```

There are several elegant schemes for global transformations of this form.

Constant propagation is most useful for detecting cases in which a mutant is not equivalent to the subject program; any change which can affect the known value of a variable can be detected in this fashion. The mechanism for testing equivalence of mutants using constant propagation is to compare at all points after the mutation site the constants which are globally propagated through the program. If they differ it is likely that the programs are not equivalent. The test is certain if there is a RETURN, HALT or some other exit statement in which the set of associated constants contains an output variable and if there is a path from the entry point of the program to the exit point. This is resolvable by dead code detection.

**Invariant Propagation:** Invariant propagation generalizes constant propagation by associating with each statement a set of invariant relations between data elements (e.g., X<0 or B=1). Although invariant propagation has met with limited applicability in compiler design, it is a powerful technique for detecting equivalent mutants, particularly those involving relational mutant operators. These operators frequently affect an expression only if it has a certain relationship to 0. For example $|x|$ changes the value of x only if x<0. In the program-mutant pair

```
IF(A.LT.0)GOTO1              IF(A.LT.0)GOTO1
B=A                 ==>      B=ABS(A)
```

the conditional allows us to determine the invariant (A>0) and this allows us to determine that the program and its mutant are equivalent since the absolute value of a positive number is that

Consider the mutation

A=B+C (partition = A;B+C) ==> A=B-C (partition = A;B-C)

Comparing the partitions shows that A has a different value in the two programs.

The same ideas are used to show equivalence. If a mutation has changed part of expression E to an expression E' but E and E' are in the same equivalence class, then the mutant is equivalent.

**Loop Invariants:** Another common transformation removes code from inside loops if the execution of that code does not depend on the iteration of the loop. Since many mutations change the boundaries of loops techniques for recognizing this invariance is useful for detecting equivalent mutants. In those cases where the mutation either increases or decreases the code within a loop, loop invariant recognition can be used to decide whether or not the effect of the loop is changed. In the following mutation, excess code is brought within the scope of the DO statement.

```
      DO 1 I=1,10      ==>      DO 2 I=1,10
      A(I)=0                    A(I)=0
1     CONTINUE             1    CONTINUE
2     B=0                  2    B=0
```

Since the assignment B=0 is loop invariant, it does not matter how many times it is executed.

**Hoisting and Sinking:** Hoisting and sinking is a form of code removal from loops in which code which will be repeatedly executed is moved to a point where it will be executed only once; this is accomplished by a calculus which gives strict conditions on when a block of code can be moved up (hoisted) or down (sunk).

The applications for equivalence testing are similar to the applications for loop invariants. The major difference is that hoisting and sinking applies to cases in which code is included or excluded along an execution path by branching changes. These are the sorts of changes obtained by GOTO replacement and statement deletion mutations. In these cases, we get equivalence if the added or deleted code can be hoisted or sunk out of the block involved in the addition or deletion.

An example will illustrate.

```
      IF(A.EQ.0)GOTO1   ==>      IF(A.EQ.0)GOTO 2
      A=A+1                      A=A+1
2     B=0                   2    B=0
      GO TO 3                    GO TO 3
1     B=0                   1    B=0
3      .                    3     .
       .                          .
       .                          .
```

In this example B is set to 0 regardless of whether it is assigned its value at line 1 or at line 2. The assignment to B can be hoisted as follows:

```
      B=0
      IF(A.EQ.0)GO TO 3
      A=A+1
3      .
       .
       .
```

Since both programs are thus transformed, they are equivalent.

**Dead Code:** Dead Code detection is geared toward identifying sections of code which cannot be executed or whose execution has no effect. Dead code algorithms exist for detecting several varieties of dead code situations. We have already used dead code analysis as a subproblem in the propagation problems above. Dead code analysis is also useful to directly test equivalence, particularly for those mutations arising from an alteration of control flow.

A typical application is to analyze the program flowgraphs. If, for example, a mutation disconnects the graph and neither connected component consists entirely of dead statements, then the mutant cannot be equivalent. Such disconnection is possible by the mutant which inserts RETURNs in Fortran subroutines.

Another common situation involves applying mutations to sites in a program which are themselves dead code; this is the classical compiler code optimization problem: we must detect dead code since any mutations applied to it are equivalent.

Dead code analysis can also be used to show nonequivalence by using it to demonstrate that a mutation has "killed" a block of code.

**Postprocessing the Mutants:** Optimizing transformations can be implemented as a postprocessor to a mutation system. User experience is that it is relatively easy to kill as may as 90% of the live mutants. To the remaining 10%, an equivalence heuristic such as the rules sketched above can be applied.

The difficulty of judging equivalent mutants from those remaining after the postprocessing stage both helps and hinders the testing process. On one hand, forcing testers and programmers to "sign off" on equivalent mutants enforces a unique sort of accountability in the testing phase of program development. On the other hand, particularly clever programming leads to many equivalent mutants whose equivalence is rather a nuisance to judge; carelessness for these programs may lead to error proneness. Our experience, however, is that production programs present no special difficulties in this regard.

## Bibliographic Notes

Detecting mutant equivalences is inherent in mutation testing, and the problem was described in [DeMillo, 1978a] and [DeMillo, 1979a]. Acree's thesis presents a discussion of the experiments used to evaluate human equivalence detection [Acree, 1980]. Baldwin and Sayward [Baldwin, 1979] noticed the relationship between mutant equivalence and optimization. These algorithms also appear in [Acree, 1979]. Tanaka [Tanaka, 1981] designed and implemented an equivalence checking post processor which uses some of the data flow analysis techniques described in this chapter.

# Chapter 8

# Error Detection

A program testing technique serves two purposes. It raises the user's confidence that a correct program is really correct. The other major function of program testing is to detect errors in programs that are not correct. In Chapter 6, we saw a number of instances in which program mutation is capable of detecting the presence of errors -- even when other techniques fail to do so. Recall that a testing technique is <u>reliable</u> if it always detects errors of a certain type. Much current research in program testing centers on developing test techniques which are reliable for classes of errors. Our goal in this chapter will be to examine program mutation in comparison with other well studied reliable test methodologies. We will describe a number of error types and show by example how the mutant operators desribed in Chapters 2 and 4

## Simple Errors

If the program contains a simple error (i.e., one represented by an error operator), then one of the mutants generated by the system will be correct. The error will be discovered when an attempt is made to eliminate the correct program since its behavior will be correct but the progam being tested will give differing results. If the program contains simple k-order errors the errors will also be detected (see Chapter 11 for an example).

## Dead Statements

Many programming errors manifest themselves in "dead code", that is, source statements that are unexecutable or, more seriously, give incorrect results regardless of the data presented. Such errors may persist for weeks or even years if the errors lie in rarely executed portions of the program.

Therefore, a reasonable first goal in testing a program is to insist that each statement be executed at least once. Typical methods for achieving this goal include, for example, the insertion of instruction counters into straight line segments of the program, so that a non-zero vector of counters indicates that the instrumented statements have all been executed at least once.

During mutation analysis, the goal outlined above will be viewed from a slightly different perspective. If a statement cannot be executed, then clearly we can change the statement in any way we want, and the effects of the changes will not be noticeable as the program runs -- in particular the altered program will not be distinguishable in its output behavior from the original one. There is, however, a mutant operator which draws the tester's attention to this situation in a more economical way. Among the mutants are those which replace in turn the first statement of every basic block by a call to a routine which aborts the run when it is executed. Such mutations are extremely unstable since any data which causes the execution of the replaced statement will also cause the mutant to produce incorrect results and hence to be eliminated. The converse is also true. That is, if any of these mutants survives the analysis then the altered statement has never been executed.

Therefore, accounting for the survival of these mutants gives important information about which sections of the program have been executed.

This analysis shows why apparently useful testing heuristics can lead one astray. For example, it has been suggested that not executing a statement is equivalent to deleting it, but this discussion shows how such a strategy can fail. A statement can be executed and still serve no useful purpose. Suppose that we replace every statement by a convenient NO-OP such as the Fortran CONTINUE. The survival or elimination of such mutants gives more information than merely whether or not the statement has been executed. It indicates whether or not the statement has any observable effect upon the output. If a statement can be replaced by a NO-OP with no observable effect, then it can indicate at best that machine time is wasted in its execution (possibly a design error) and very often a much more serious error.

Insuring that every statement is executable is no guarantee of correctness. Predicate errors or coincidental correctness may pass undetected even if every statement is successfully executed. We will return to these error types later in this Chapter.

## Dead Branches

An improvement over simply analyzing the execution of statements can be had by analyzing the execution of branches, attempting to execute every branch at least once.

Consider the program segment

```
A;
IF(<expression>) THEN B;
C;
```

All statements A,B and C can be executed by a single test case. It is not true however that in this case all branches have been executed. In this example the empty else clause branch can be bypassed even though A,B and C are executed.

However, the requirement that every branch be traversed can be restated: every predicate must evaluate to both TRUE and FALSE. The latter formulation is used in mutation analysis. The mutant operators **trueop** and **falseop** replace each logical expression by Boolean constants. Like the statement analysis mutations described above, these mutations tend to be unstable and are easily eliminated by almost any data. If these mutants survive, they point directly to a weakness in the test data which might shield a possible error.

Mutating each relation or each logical expression independently actually achieves a stronger test than that achieved by the usual techniques of branch analysis. For consider the compound predicate

```
IF(A.LE.B.AND.C.LE.D)THEN ...
```

Simple branch coverage requires only two test cases to test the predicate. But suppose that the test points for the covering test are $A < B \wedge C < D$ and $A < B \wedge C > D$.

These points have the effect of only testing the second clause. This kind of analysis fails to take into account the hidden paths implicit in compound predicates. In testing all the hidden paths, program mutation requires at least three points to test the predicate, corresponding to the branches $(A>B,C>D)$, $(A<B,C>D)$, and $(A\leq B,C\leq D)$.

As a more concrete example, consider the program shown in Figure 1 (cf. Program B4). It is intended to calculate the number of days between two given dates. The predicate which determines whether a year is a leap year is incorrect. Notice that if the year is divisible by 400 (i.e., if year REM 400 = 0) it is necessarily divisible by 100 (ie, year REM 100 = 0). Therefore, the logical expression formed by the conjunction of these clauses is equivalent to the second clause alone. Alternatively the expression year REM 100 = 0 can be replaced by the logical constant TRUE and the resulting mutant is equivalent to the original program. Since it is not obvious what the programmer had in mind, the error is discovered. Mutation analysis also shows that the assignment daysin(12):=31 is redundant and can be removed from the program.

```
PROCEDURE calendar(INTEGER VALUE day1,month1,day2,month2,year);
 BEGIN
  INTEGER days
  IF month2=month1 THEN days=days2-days1
    COMMENT if the dates are in the same month, then
            we can compute the number of days directly;
  ELSE
   BEGIN
    INTEGER ARRAY daysin(1..12)
    daysin(1):=31;daysin(3):=31;daysin(4):=30;
    daysin(5):=31;daysin(6):=30;daysin(7):=31;
    daysin(8):=31;daysin(9):=30;daysin(10):=31;
    daysin(11):=30;daysin(12):=31;
    IF ((year REM 400)=0) OR
      ((year REM 100)=0 and (year REM 400)=0)
      THEN daysin(2):=28 ELSE daysin(2):=29;
    COMMENT set daysin(2) according to whether or not
            year is leap year;
    days:=day2+(daysin(month1)-day1);
    COMMENT this yields the number of days in complete
            intervening months;
    FOR i:=month1 +1 UNTIL month2-1 DO days:=daysin(i)+days;
    COMMENT add in the days in complete months;
   END
   WRITE(days)
 END;
```

Figure 1.

## Data Flow Errors.

A program may access a variable in one of three ways. A variable is said to be <u>defined</u> if the result of a statement is to assign a value to the variable. A variable is said to be <u>referenced</u> if its value is required by the execution of a statement. Finally, a variable is said to be <u>undefined</u> if the semantics of the language does not explicitly give any other value to the variable. Examples of undefined variables are the values of local storage after procedure return or Fortran DO loop indices after normal loop termination.

We define three types of data flow anomalies which are often indicative of program errors. These anomalies are consecutive accesses to a variable of the following forms:

1. undefined then referenced,
2. defined then undefined,
3. defined then redefined.

Anomaly 1 is almost always indicative of an error, even if it occurs only on a single path between the point at which the variable becomes undefined and its point of reference. Anomalies 2 and 3 tend to indicate errors when they are unavoidable, that is, when they occur along every control path.

The second and third types of anomalies are attacked directly by mutation operators. If a variable is defined and is not used then in most cases the defining statement can be eliminated without effect (by insertion of a CONTINUE statement for instance). This may not be the case if in the course of defining the variable a function with side effects is invoked. In this case, the definition can very likely be altered in many ways with no effect on the side effect, resulting in the variable being given different values. An attempt to remove these mutations will usually result in the anomaly being discovered.

It is more difficult to see which operators address anomalies of the first type; the underlying errors are attacked by the discipline imposed by program mutation. A tester creates and executes mutants in a specific test environment: a large interpretive system. Whenever the value of a variable becomes undefined, it is set by the interpreter to the unique constant UNDEFINED. Before every variable reference, a check is performed by the interpreter to see if the variable has undefined values. If the variable is UNDEFINED the error is reported to the user, who can then take action. Several examples of error detection by the interpreter are presented in Chapter 6.

**Domain Errors.**

A domain error occurs when an input value causes an incorrect path to be executed due to an error in a control statement. Domain errors are to be contrasted with computation errors which occur when an input value causes the correct path to be followed but an incorrect function of the input value is computed along that path due to an error in a computation statement. These notions are not precise and it is difficult with many errors to decide in which category they belong (cf. the error classifications in Chapter 2).

For a program containing N input variables (e.g., parameters, arrays, and I/O variables), any predicate in the program can be treated algebraically and can thus be described by a surface in the N dimensional input space. If, as often happens, the predicate is linear, then the surface is a hyperplane.

Consider a two dimensional example with input variables I and J: $I+2J \leq -3$. The domain strategy tests this predicate using three test points, two on the line I+2J=3, and one point which lies off the line, but within an envelope of width 2d centered on the line . Call these points A,B and C (see Figure 2). If A,B, and C yield correct output, then the defining curve of the predicate must cut the sections of the triangle ABC. Choosing d small enough makes the chance of the predicate actually being one of these alternatives small. Therefore, we have gained some confidence that the predicate is correct.

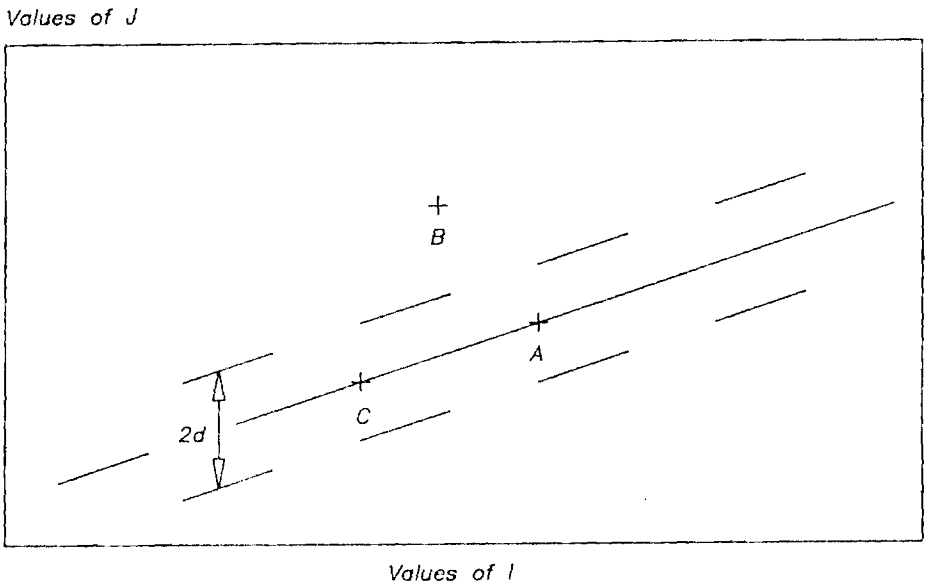



Figure 2.
Domains for I+2J < 3

Program mutation also deals with the issue of domain errors. Indeed the domain strategy can be implemented using mutation once a simple observation is made: it is not necessary that points A and B both lie on the line — it is only necessary that the line separate them or that they do not both lie on the same side of the line. Hereafter, we will work with the domain stategy using this simplifying assumption.

There are three error operators which generate mutants causing the tester to generate the required points. Intuitively, we can think of the mutations as posing certain alternatives to the predicate in question. These alternatives require the tester to supply "reasons" (in the form of test data) why the alternative predicate cannot be used in place of the original.

**Relational Operator Replacement.** Changing an inequality operator to a strict inequality, weakening the operator, or changing its sense generates a mutant which can only be eliminated by a test point which exactly satisfies the predicate. For example changing $I+2J\leq3$ to $I+2J<3$ requires the tester to generate a point on the line $I+2J=3$ which satisfies the first predicate but which does not satisfy the second predicate.

**Twiddle.** Recall from Chapter 2 that twiddle is a unary operator denoted by ++ or --, depending on its sense. Usually ++a is defined to be a+1 if a is an integer and a+.01, if a is real. In some cases ++a is defined to be sensitive to the magnitude of a. The complementary operator --a is defined similarly.

Graphically, the effect of twiddle is to move the proposed constraint a small distance from the original line. In order to eliminate these mutants, a data point must be found which satisfies one constraint but not the other and is hence very close to the original line.

**Other Replacements.** These operators replace data references with other syntactically meaningful data references and similarly for operators. These effects are related to the phenomenon of "spoilers" which are described later in this chapter.

Replacements are the main source of complexity in the mutation process, since the number of data substitution mutant alone grows approximately quadratically in the size of the program being tested (see Chapter 5). The practical effect of considering so many alternatives is to increase the total number of data points necessary for their elimination. This leads by the domain strategy to an increased confidence that the predicate has been correctly chosen.

For comparison, let us work through the program in Figure 3. No specifications are given for this program, but the program can be compared against a presumably correct version; in any case the program is useful since it involves only two input variables.

```
READ I,J;
IF I<J+1
   THEN K=I+J-1
   ELSE K=2*I+1;
IF K≥I+1
   THEN L=I+1
   ELSE L=J-1;
IF I=5
   THEN M=2*L+K;
   ELSE M=L+2*K-1
WRITE M;
```

Figure 3.

The program has only three predicates:

$I \leq J+1$, $K > I+1$, and $I=5$.

The effect of changing the first of these is typical, so we will deal with it.

Figure 4 is a listing of all the alternatives tried for the predicate I<J+1. Some of these are redundant (e.g., ++I<J+1 and I≤--J+1), but this is merely an artifact of the generation device; the redundancies can be easily removed. The alternative predicates introduced in this way are illustrated in Figure 5. The original predicate line is the heavy line. It has been suggested that the program of Figure 3 contains the errors shown in Table 1.

| statement/expression | should be |
|---|---|
| K≥I+1 | K≥I+2 |
| I=5 | I=5-J |
| L=J-1 | L=I-2 |
| K=I+J-1 | THEN IF(2*J<-5*I-40)<br>THEN K=3;<br>ELSE K=I+J-1; |

Table 1. Domain Errors

We leave it to the reader to verify that attempting to eliminate the alternative K≥I+2 necessarily ends with the discovery of the first error. Note that this is not trivial since errors 1 and 4 can interact in a subtle way. In the sequel we show how the remaining errors are dealt with.

1. IF(I<J)
2. IF(I≤J+2)
3. IF(I≤J+1)
4. IF(I≤J+J)
5. IF(1≤J+1)
6. IF(2≤J+1)
7. IF(5≤J+1)
8. IF(I≤1+1)
9. IF(I<2+1)
10. IF(I≤5+1)
11. IF(I<J+5)
12. IF(-I≤J+1)
13. IF(++I≤J+1)
14. IF(--I≤J+1)
15. IF(I<-J+1)
16. IF(I≤++J+1)
17. IF(I<--J+1)
18. IF(I≤-(J+1))
19. IF(I<J-1)
20. IF(I≤MOD(J,1))
21. IF(I<J)
22. IF(I≤1)
23. IF(I<J+1)
24. IF(I=J+1)
25. IF(.NOT.I=J+1)
26. IF(I>J+1)
27. IF(I≥J+1)

Figure 4.

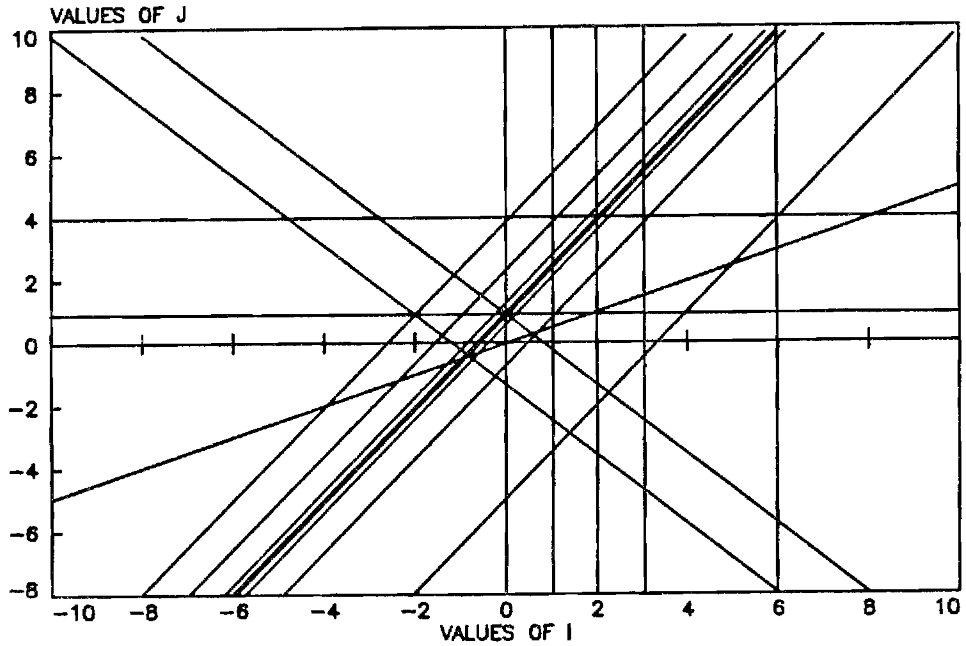



Figure 5.
Alternative Predicate Domains

The introduction of the unary ++ and -- operators can be generalized in several useful ways. In addition to the twiddle operators, we consider the unary operator - and the operators ABS (absolute value), -ABS (negative absolute value), and ZPUSH (zero push). Consider the statement A=B+C. In order to eliminate the mutants A= ABS(B)+C, A=B+ABS(C), and A=ABS(B+C), we must generate a set of test points in which B is negative (so that B+C differs from ABS(B+C), C is negative, and B+C is negative). Notice that if it is impossible for B to be negative then this is an equivalent mutation. In this case, the proliferation of these alternatives can either be a nuisance or an important documentation aid, depending upon the testers' point of view. The topic of equivalent mutants will be taken up again later.

In similar fashion, negative absolute value insertion forces the test data to be positive. We use the term domain pushing for this process. By analogy to the domain strategy, these mutations push the tester into producing test cases where the domains satisfy the given requirements.

Zero Push is an operator defined so that ZPUSH(x) is x if x is nonzero, and otherwise is undefined so that the mutant dies immediately. Hence the elimination of this mutant requires a test point in which the expression x has the value zero.

Applying this process at every point where an absolute value sign can be inserted gives a scattering effect. The tester is forced to include test cases acting in various positions in several problem domains. Very often, in the presence of an error, this scattering effect causes a test case to be generated in which the error is explicit.

Returning to the example in Figure 3, we can generate the additional alternatives shown in Figure 6. Figure 7 shows the domains into which these mutants push. Even this simple example generates a large number of requirements!

```
1. IF(ABS(I)>J+1)
2. IF(I>ABS(J)+1)
3. IF(I>ABS(J+1))
4. K=(ABS(I)+J)-1
5. K=(I+ABS(J))-1
6. K=ABS(I+J)-1
7. K=ABS((I+J)-1)
8. K=2*ABS(I)+1
9. K=ABS(2*I)+1
10. K=ABS(2*I+1)
11. IF(ABS(K)<I+1)
12. IF(K<ABS(I)+1)
13. IF(K<ABS(I+1))
14. L=ABS(I)+1
15. L=ABS(I+1)
16. L=ABS(J)-1
17. L=ABS(J-1)
18. IF(.NOT.ABS(I)=5)
19. M=2*ABS(L)+K
20. M=2*L+ABS(K)
21. M=ABS(2*L+K)
22. M=ABS(L)+2*K-1
23. M=L+2*ABS(K)-1
24. M=ABS(L+2*K)-1
25. M=ABS(L+2*K-1)
```

Figure 6.

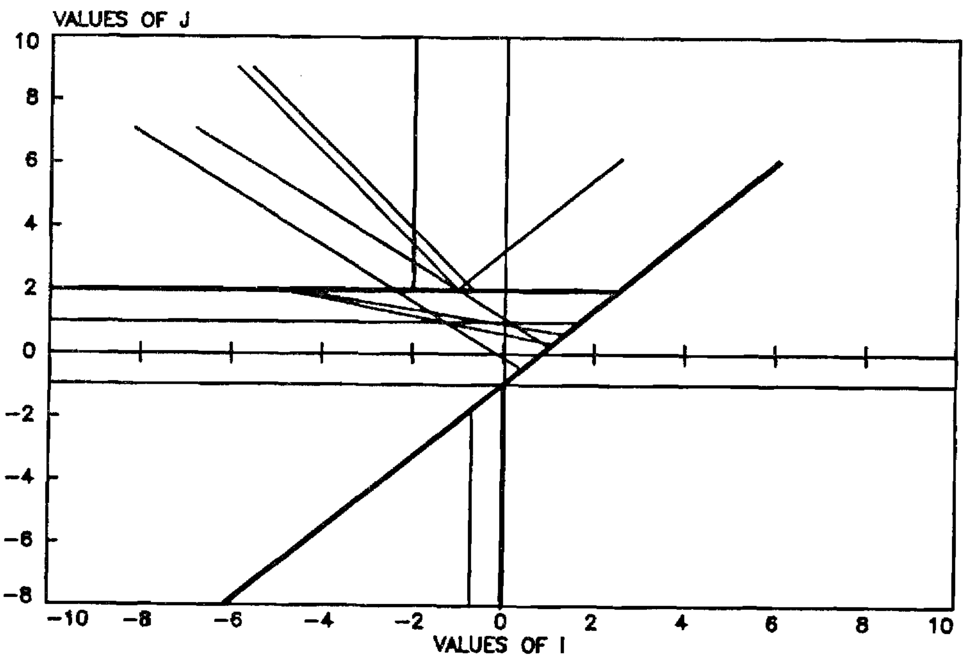



Figure 7.
Effects of Domain Pushing

One effect of the error L=J-1 is that <u>any</u> test point in the area bounded by I=J+1 and I=1 will return an incorrect result. But this is precisely the area that mutants 8,9, and 10 push us into. So, the error could not have gone undiscovered in mutation analysis.

This process of pushing the tester into producing data satisfying some criterion is also often accomplished by other mutations. Consider the program in Figure 8, which is based on a text reformatter program and which is also discussed in Appendix B (Program B11).

```
alarm:=FALSE
bufpos:=0;
fill:=0;
REPEAT
 incharacter(cw);
 IF cw=BL or cw=NL THEN
    IF fill+bufpos ≤ maxpos THEN
           outcharacter(BL);
    ELSE
     BEGIN
      outcharacter(NL);
      fill:=0;
      FOR k:=1 STEP 1 UNTIL bufpos DO outcharacter(buffer[k])
      fill:=fill+bufpos;
      bufpos:=0
     END
   ELSE
    IF bufpos = maxpos THEN alarm:=TRUE;
    ELSE BEGIN
     bufpos:=bufpos+1;
     buffer[bufpos]:=cw
    END
UNTIL alarm or cw=ET
```

Figure 8.

Consider the mutant which replaces the first statement fill:=0 with the statement fill:=1. The effect of this mutation is to force a test case to be defined in which the first word is less than maxpos characters long. This test case then detects one of the five errors originally reported in Appedix B. The surprising thing is that the effect of this mutation seems to be totally unrelated to the statement in which the mutation takes place!

## Special Values

Another form of test which has been studied is special values testing. Testing of special values is defined in terms of a number of "rules". For example:

1. Every subexpression should be tested on at least one test case which forces the expression to be zero.

2. Every variable and every subexpression should take on a distinct set of values in the test case.

The relationship between the first rule and domain pushing (via zero values mutations) has already been discussed. The second rule is undeniably important. If two variables are always given the same value then they do not act as free variables and a reference to the first can be uniformly replaced with a reference to the second. But this is also an error operator and the existence of these mutations enforces the goals of Rule 2.

A slightly more general method of enforcing Rule 2 might use the following device. A special array exactly as large as the number of subexpressions to be computed in the program is kept. Each entry in this array has two additional tag bits which are intialized to their low values indicating that the array is uninitialized. As each subexpression is encountered in turn, the value at that point is recorded in the array and the first tag bit is set. Subsequently, when the subexpression is again encountered if the second tag is still off the current value of the expression is compared against the recorded value. If these values differ the second tag is set to high values; otherwise no change is made. By counting those expressions in which the second tag bit is low and the first is high one can infer which expressions have not had their values altered over the test case. Mutations could be constructed to reveal this.

## Coincidental Correctness

The result of evaluating a given test point is <u>coincidentally correct</u> if the result matches the intended value in spite of a computation error. For example, if all our test data results in the variable I taking on the values 2 and 0, then the computation J=I*2 may be coincidentally correct if the intended calculation was J=I**2.

The problem of coincidental correctness is central to program testing. Every programmer who tests an incorrect program and fails to find the errors has really encountered an instance of coincidental correctness. In spite of this, there has been no direct assault on the problem and some authors have gone so far as to say that the problems of coincidental correctness are intractable.

In mutation analysis, coincidental correctness is attacked by by the use of <u>spoilers</u>. Spoilers implicitly remove from consideration data points for which the results could obviously be coincidentally correct — this "spoils" those data points. For example by explicitly creating the mutation

J=I*2 ==> J=I**2,

we spoil those test cases for which I=0 or I=2 are coincidentally correct and require that at lest one test case have an alternative value.

Continuing with the example of Figure 3, Figure 9 shows the spoilers and their effects associated with the statement M=L+2*K-1. Notice that a single spoiler may be associated with up to four different lines depending on the outcome of the first two predicates in the program. In geometric terms (see Figure 11), the effects of the spoilers are that within each data domain for each line there must be at least one test case which does not lie on the given line. In broad terms, the effects of this are to require that a large number of data points for which the possibilities of coincidental correctness are very slight.

1. M=(L+1*K)-1
2. M=(L+3*K)-1
3. M=(I+2*K)-1
4. M=(J+2*K)-1
5. M=(K+2*K)-1
6. M=(L+2*J)-1
7. M=(L+2*I)-1
8. M=(L+2*L)-1
9. M=(L+I*K)-1
10. M=(L+J*K)-1
11. M=(L+K*K)-1
12. M=(L+L*K)-1
13. M=(L+2*K)-I
14. M=(L+2*K)-J
15. M=(L+2*K)-K
16. M=(L+2*K)-L
17. M=(1+2*K)-1
18. M=(2+2*K)-1
19. M=(5+2*K)-1
20. M=(L+2*1)-1
21. M=(L+2*2)-1
22. M=(L+2*5)-1
23. M=(L+5*K)-1
24. M=(-L+2*K)-1
25. M=(L+-2*K)-1
26. M=(L+2*-K)-1
27. M=(L+2*--K)-1
28. M=(L+2*K)-1
29. M=((L+2*K)-1)
30. M=(L+2+K)-1
31. M=(L+2-K)-1
32. M=(L+MOD(2,K))-1
33. M=(L+2/K)-1
34. M=(L+2**K)-1
35. M=(L+2)-1
36. M=(L+K)-1
37. M=L-2*K-1
38. M=(MOD(L,2*K))-1
39. M=L/2*K-1
40. M=L*2*K-1
41. M=L**(2*K)-1
42. M=L-1
43. M=(2*K)-1
44. M=L+2+K+1
45. M=MOD(L+2*K,1)
46. M=(L+2*K)/1
47. M=(L+2*K)*1
48. M=(L+2*K)**1
49. M=(L+2*K)
50. M=1

Figure 10

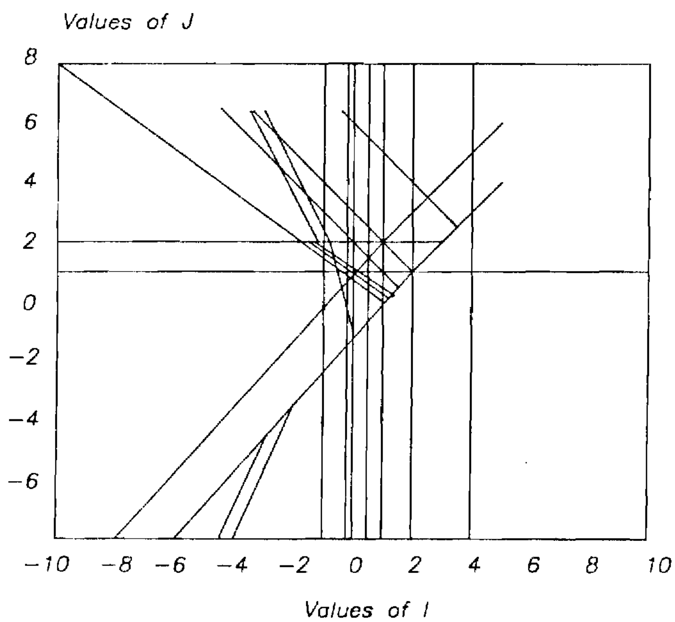



Figure 11.
Effects of Spoilers

Often the fact that two expressions are coincidentally the same over the input data is a sign of a program error or of poor testing. The sorting program of Figure 12 is described in Appendix B (Program B2), and it performs correctly for a large number of input values. If, however, the statements following the IF statement are never executed for some loop iteration it is possible for R3 to be incorrectly set and an incorrectly sorted array will result.

By constructing the mutant which replaces the statement

```
a(R1):=R0 ==> a(R1):=a(R3)
```

it is clear that there are two ways of defining R0, only one of which is used in the test data. This exposes the error.

```
FOR R1=0 BY 1 TO N BEGIN
  R0:=a(R1);
  FOR R2=R1+1 BY 1 TO N BEGIN
   IF a(R2)>R0 THEN BEGIN
    R0:=a(R2);
    R3:=R2
   END
  END
  R2:=R0;
  a(R1):=R0;
  a(R3):=R2
 END;
```

Figure 12.

## Missing Path Errors

A program contains a missing path error if a predicate is required which does not appear in the subject program, causing some data to be computed by the same function when an altogether different function of the input data is called for. Such missing predicates can really be the result of two different problems, however, so we might consider the following alternative definitions.

A program contains a *specificational missing path error* if two cases which are treated differently in the specifications are incorrectly combined into a single function in the program. On the other hand, a program contains a *computational missing path error* if within the domain of a single specification a path is missing which is required only because of the nature of the algorithm or of the data involved.

An example of a specificational error is the fourth error from Table 1. Although this error might result from a specification there is nothing in the code itself which could give any hint that the data in the range $2*J < 5*I-40$ is to be handled any differently than shown in the program.

As an example of the second class of path error consider the subroutine shown in Figure 13. The input consists of a sorted table of numbers and an element which may or may not be in the table. The only specification is that upon return $X(LOW) \leq A \leq X(HIGH)$ and $HIGH \leq LOW+1$. A problem arises if the program is presented with a table of only one entry, in which case the program diverges.

In the specifications there is no clue that a one-entry table is to be treated any differently from a $k>1$ entry table. The algorithm makes it a special case.

```
         SUBROUTINE BIN(X,N,A,LOW,HIGH)
         INTEGER X(N),N,A,LOW,HIGH
         INTEGER MID
         LOW=1
         HIGH=N
6        IF(HIGH-LOW-1)7,12,7
12       RETURN
7        MID=(LOW+HIGH)/2
         IF(A-X(MID))9,10,10
9        HIGH=MID
         GO TO 6
10       LOW=MID
         GO TO 6
         END
```

Figure 13.

Computational missing path problems are usually caused by requirements to treat certain values (e.g., negative numbers) differently from others. When this occurs, data pushing and spoiling often lead to the detection of the errors. In the example under consideration here an attempt to kill either of the mutants

```
IF(HIGH-LOW-1)12,12,7
```

or

```
MID=(LOW+HIGH)-2
```

will cause us to generate a test case with a single element.

Since mutation analysis — like all testing techniques -- deals mainly with the program under test, the problem of dealing with specificational missing path errors appears to be considerably more difficult. Under the Competent Programmer Assumption and the coupling effect, however, a tester who has access to an "oracle" for the program specifications can assume that the mutants cover <u>all</u> program behavior! So by consulting the specifications the tester can detect missing paths by noting incomplete behavior and thus uncover any missing paths. But since the assumptions of a competent programmer and coupling are statistical and since it may be infeasible to check for incomplete behavior, the chances of detecting such missing paths are not certain.

To see this failure, consider the missing path error discussed above (the fourth error in Table 1). It is possible to generate test data which is adequate but which fails to detect the missing path error because there is no oracle to consult for completeness of behavior. This appears to be a fundamental limitation of the testing process. Unlike, say, program verification, program testing does not require uniform a priori specifications; rather we only ask that the tester be able to judge correctness on a case-by-case basis. It is our view that the only way to attack these problems is to start with a core of test cases generated from specifications, independent of the subject program. This core of test cases can then be augmented to achieve stronger goals.

## Missing Statement Errors

By analogy with missing path errors, a missing statement error is defined by a statement which should appear in the program but which does not. It is not clear that the techniques of statement analysis can be used to uncover these errors. In fact, it is rather surprising that program mutation — a technique which is directly oriented toward examining the effect of a modification to a statement — can be used to detect missing statements at all!

To see how this can be accomplished, consider the program shown in Figure 14. This program accepts a vector V of length N and returns in MPSUM the value

V(i)+V(i+1)+...+V(N)

where j=i-1 is the smallest index such that V(j) is strictly positive. In degenerate cases, MPSUM=0 is returned.

There is a missing RETURN statement which should follow the IF statement. The effect of the error is to cause undefined behavior when the vector V is uniformly nonpositive (undefined, since DO loop variables are of indeterminate value after normal completion of the loop).

A simple mutation of MPADD is the transformation

```
DO 1 I=1,N ==> DO 1 I=1,N+1.
```

This mutant fails only when the loop executes N+1 times. In this case all elements of V are nonpositive and the original program fails, so eliminating this mutant uncovers the error. But even after adding the return statement, MPADD will still be incorrect due to a missing path error. We leave it to the reader to discover the error by considering the mutant

```
DO 1 I=1,N ==> DO 1 I=1,N-1.
```

```
      SUBROUTINE MPADD(V,N,MPSUM)
      INTEGER V(N),N,MPSUM
      MPSUM = 0
      DO 1 I=1,N
1     IF(V(I).GT.0)GO TO 2
2     M=I+1
      DO 3 I=M,N
3     MPSUM=MPSUM+V(I)
      RETURN
      END
```

Figure 14.

## Bibliographic Notes

The usefulness of program mutation for detecting errors was pointed out by DeMillo, Lipton and Sayward in [DeMillo, 1978a]. However, the first systematic investigation of classes of errors that are revealed by mutant operators was given in [Acree, 1979]. These techniques are several others which are useful in uncovering known error classes also appear in Budd's thesis [Budd, 1980].

# Chapter 9

## Field Studies

In spite of extensive theoretical and experimental analysis, systematic program testing in production programming environments is rare. Most published accounts of testing experience in large scale development efforts concentrate on ad hoc techniques which have been tailored to the parent project. On the other hand, published descriptions of systematic testing research use example programs which are small, theoretically interesting and easily adaptable to expository accounts. This leaves open the question of whether <u>any</u> systematic testing strategy can be economically applied in production programming situations. This chapter describes several field experiments with production programs of varying size and complexity.

The common thread in all of these case studies is that the programs being tested are not known beforehand to be "testable" by any technique. The programs are neither appealing nor known to be correct. In fact several of the programs were known to contain resistant errors that had escaped all of the usual debugging techniques. Other programs had been thoroughly tested by other organizations and fielded with errors that surfaced only during subsequent operation.

The programs below were tested using Fortran and Cobol mutation analyzers based on the design principles presented in Chapter 4. The test environments varied. The Fortran analyzers were implemented on a large Digital Equipment System/20. The Cobol analyzer was implemented on PR1ME Computer Corporation's 400 and 500 series computers. The level of skill of the testers also varied.

In one instance, the testers were expert mutation analyzer users. In another, the testers were unknown, and program mutation was used to evaluate the results of an independent testing effort. Although these studies used considerable machine resources, the principle bottleneck in the testing process was the human tester. In only one instance (the testing of a 2,500 statement Cobol program) did the test team have to wait appreciable lengths of time to receive the test results. On the average, expert testers were able to fully test (i.e., develop adequate test sets, correct errors discovered, and retest the modified programs) production code at the rate of 1,500 delivered source lines per tester per week.

## Mutation on Mutation

The Fortran programs which we will discuss below are key routines of a Cobol mutation analyzer whose design parallels the organization suggested in Chapter 4. These programs were tested in nearly the same form as the programs which would eventually be integrated into the operational system. The few modifications that had to be made to allow testing on a Fortran analyzer were mainly to due to operating system dependencies that were not supported in the test environments.

### NXTLIV

This program is a routine called NXTLIV. It is a key routine in the Cobol mutation analyzer and at the time of testing was known to contain an error that could not be located by the usual debugging techniques.

NXTLIV accepts as input the identifying number of a mutant of a given type and returns the number of the next live mutant, as indicated by bit maps of the live mutants. The bit maps are, in general, too large to fit in an internal array so they must be paged from a random access disk file as needed. Similar maps of the dead mutants and equivalent mutants are also stored. The program is shown below.

```
      SUBROUTINE NXTLIV(MTYPE,MUTNO)
C FIND THE NEXT LIVE MUTANT AFTER THE MUTNOth OF TYPE MTYPE
C RETURN THIS VALUE IN MUTNO.
C A VALUE OF ZERO RETURNED MEANS NO MUTANTS OF THAT TYPE
  REMAIN ALIVE.
      NOLIST
$INSERT ICS057>CPMS.COMPAR>SYSTEM.PAR
$INSERT ICS057>CPMS.COMPAR>MACHINE.SIZES.PAR
$INSERT ICS057>CPMS.COMPAR>FILENM.COM
$INSERT ICS057>CPMS.COMPAR>TSTDAT.COM
$INSERT ICS057>CPMS.COMPAR>MSBUF.COM
      LIST
      INTEGER MTYPE,MUTNO
      INTEGER I,J,K,L,WORD,BIT
      LOGICAL ERR
C     CALL TIMER1(33)
C ASSUME THAT THE RECORD CONTAINING THE LIVE BIT MAPS FOR
C MUTNO IS ALREADY PRESENT, UNLESS MUTNO=0.
      K=BPW-1
C CHECK TO SEE IF WE ARE AT THE END OF A PHYSICAL RECORD
      IF(MUTNO.EQ.0)TO TO 1
      IF(MOD(MUTNO,K*MSFRS).EQ.0)GO TO 24
      GO TO 10
1     CALL REARAN(MSFILE,LIVBUF,MSFRS,LIVPTR,ERR)
      IF(ERR)CALL ABORT('(NXTLIV) ERROR IN MUTANT STATUS FILE',36)
     CALL REARAN(MSFILE,EQUBUF,MSFRS,EQUPTR,ERR)
      IF(ERR)CALL ABORT('(NXTLIV) ERROR IN MUTANT STATUS FILE',36)
      CALL REARAN(MSFILE,DEDBUF,MSFRS,DEDPTR,ERR)
      IF(ERR)CALL ABORT('(NXTLIV) ERROR IN MUTANT STATUS FILE',36)
      CHANGD=.FALSE.
      WORD=1
      BIT=2
      GO TO 20
10    WORD=MOD((MUTNO)/(K),MSFRS)+1.
      BIT=MOD(MUTNO,K)+2
20    DO 22 J=WORD,MSFRS
      L=LIVBUF(J)
      IF(L.NE.0)GO TO 23
```

An error has been detected; the correct output for MUTNO is 13 instead of 14. This error resulted from choosing a starting point in the middle of a word of zero bits. NXTLIV ordinarily searches the bits of each word looking for the next "1", but for efficiency a whole word is compared to zero before the search is begun. If all bits are set low, MUTNO is incremented by the word length and the next word is accessed. A correct algorithm would increment MUTNO only by the number of bits left to be examined in the word. The only way this can make a difference in the original program is for NXTLIV to be called in such away as to stop at a "1" bit in the middle of the word, which is otherwise all 0's, and then by a mutant failure or equivalence (outside the routine) to have that bit turned off before NXTLIV is called again for the next mutant to be considered. Obviously this situation is so rare that it is bound to defy haphazard debugging attempts but is nonetheless common enough to cause irritation in a production-sized Cobol run.

The needed fix is to replace

```
MUTNO=MUTNO+K
```

by

```
MUTNO=MUTNO+(K-(BIT-2)).
```

After eliminating all SAN mutants and turning on the remaining error operators, a total of eleven test cases killed all but 50 of 1,514 mutants, about 96.7 percent of the total. Eventually the tester's attention was directed to the mutant at line 45

BIT=2 ==> I=2.

The test case 15 in Table 2 is an attempt to eliminate this mutant. The program again failed and another error was found. This error is also related to the test for the entire word of zeroes. By starting in the middle of a word of zeroes, the BIT pointer is not correctly set to 2 to begin searching the next word. The correction is to replace

```
      BIT=2
22    CONTINUE
```

by

```
22    BIT=2
```

An interesting note is that this "correction" is actually a mutation that the tester would have had to eliminate in any event, so in effect the error was uncovered by the coupling effect before it was explicitly considered.

The complete analysis of the corrected program required the elimination of 1,580 mutants. The corrected algorithm has since been running without known failure in an operational mutation analyzer.

**MOVENW and MOVENM**

These routines were tested using a more sophisticated mutation analyzer than the one used to test NXTLIV. Only minor modifications in the source code were required to conform to the requirements of the test environment.

The MOVENM and MOVENW routines were believed to be correct at the time of testing. The listings for MOVENW and MOVENM are shown below.

```
      SUBROUTINE MOVENW(SOURCE,SLEN,DEST,DLEN)
      INTEGER MLEN, K, SUB2, SUB1, LOOPHI, I, IHI, IER
      INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
      CHAR MEMORY(425)
      INTEGER DLEN, DEST, SLEN, SOURCE
      INPUT OUTPUT IER, MEMORY
      INPUT DLEN, DEST, SLEN, SOURCE
      MLEN = DLEN                                         1
      IF(SLEN .LT. MLEN) MLEN = SLEN                   2  3
      LOOPHI = (DEST + MLEN) - 1                          4
      SUB2 = SOURCE - 1                                   5
      DO 20 SUB1=DEST, LOOPHI                             6
      SUB2 = SUB2 + 1                                     7
      K = MEMORY(SUB2)                                    8
      IF(K .EQ. '#') IER = 4                           9 10
20    MEMORY(SUB1) = K                                   11
      IF(IER .NE. 0) GOTO 9999                        12 13
      IF(DLEN .LE. MLEN) GOTO 9999                    14 15
      I = LOOPHI + 1                                     16
      LOOPHI = (DEST + DLEN) - 1                         17
      DO 30 SUB1=I, LOOPHI                               18
30    MEMORY(SUB1) = ' '                                 19
9999  CONTINUE                                           20
      RETURN                                             21
      END

      SUBROUTINE MOVENM(SOURCE,SLEN,SDEC,DEST,DLEN,DDEC,TYPPE)
      LOGICAL NEGNO
      INTEGER X(5), PTNEGD, PTNEGS, K, SUB2, SUB1, LOOPHI, LEND
      INTEGER LENS, I, IHI, DDECPT, SDECPT, IER. STMT(3,10)
      INTEGER CODE(30), SYMTAB(10,9)
      CHAR MEMORY(425)
      INTEGER TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
      INPUT OUTPUT IER, MEMORY
      INPUT TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
      PTNEGS = (SOURCE + SLEN) - 1                       23
      PTNEGD = (DEST + DLEN) - 1                         24
      CALL UNPACK(MEMORY(PTNEGS),X,5)                    25
      NEGNO = X(2) .EQ. '-'                              26

      X(2) = ' '                                         27
      IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)         28 29
      LENS = SLEN - SDEC                                 30
      LEND = DLEN - DDEC                                 31
      SDECPT = SOURCE + LENS                             32
      DDECPT = DEST + LEND                               33
      SUB1 = DDECPT - 1                                  34
      IF(SDEC .EQ. 0 .OR. DDEC .EQ. 0) GOTO 22        35 36
      IHI = (SDEC + SDECPT) - 1                          37
      IF(DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1    38 39
      DO 20 SUB2=SDECPT, IHI                             40
      SUB1 = SUB1 + 1                                    41
      K = MEMORY(SUB2)                                   42
      IF(K .EQ. '#') IER = 4                          43 44
20    MEMORY(SUB1) = K                                   45
```

```
      IF(IER .NE. 0) GOTO 50                         46 47
22    IF(DDEC .LE. SDEC) GOTO 30                     48 49
      I = SUB1 + 1                                      50
      IHI = (DEST + DLEN) - 1                           51
      DO 25 SUB1=I, IHI                                 52
25    MEMORY(SUB1) = '0'                                53
30    LOOPHI = LEND                                     54
      IF(LENS .LE. LEND) LOOPHI = LENS               55 56
      SUB1 = DDECPT                                     57
      SUB2 = SDECPT                                     58
      IF(LEND .EQ. 0) GOTO 50                        59 60
      IF(LENS .EQ. 0) GOTO 41                        61 62
      DO 40 I=1, LOOPHI                                 63
      SUB1 = SUB1 - 1                                   64
      SUB2 = SUB2 - 1                                   65
      K = MEMORY(SUB2)                                  66
      IF(K .EQ. '#') IER = 4                         67 68
40    MEMORY(SUB1) = K                                  69
      IF(IER .NE. 0) GOTO 50                         70 71
      IF(LEND .LE. LENS) GOTO 50                     72 73
41    IHI = SUB1 - 1                                    74
      DO 45 I=DEST, IHI                                 75
45    MEMORY(I) = '0'                                   76
50    X(2) = '-'                                        77
      IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)        78 79
      IF(.NOT. (NEGNO .AND. TYPPE .EQ. 2)) RETURN    80 81
      CALL UNPACK(MEMORY(PTNEGD),X,5)                   82
      X(2) = '-'                                        83
      CALL PACK(X,MEMORY(PTNEGD),5)                     84
      RETURN                                            85
      END
```

Program mutation on each subroutine indicated that no errors existed and that the two subroutines were correct. A listing of each subroutine with its equivalent mutants and the MUTANT STATE information is given in Appendix C.

Most of the equivalent mutants are the absolute value or ZPUSH mutants of a variable; these variables are always positive and never zero because they refer to the memory location and length for either the sending field or destination field in the Cobol MOVE statement and this cannot be negative or zero.

It is interesting to note the statement:

```
IF (K .EQ. '#') IER=4
```

This conditional is checking for undefined data. If the data is undefined, the data is moved entirely to the receiving field before the interpreter is halted and an error returned to the calling subroutine. The conditional statement:

IF (IER .NE. 0) GO TO 9999 as in MOVENW

IF (IER .NE. 0) GO TO 50 as in MOVENM

is located after the Fortran DO loop that is moving the data; if this statement were moved inside the DO loop, then the error could cause the error return before all the data is moved. The tester decided that the time to evaluate the error condition every time through the DO loop would be more time consuming than the time needed to move the remaining data to the receiving field. It should be noted that moving the undefined data to the receiving field has no effect because interpretation of the program is halted.

**MOVEED**

The MOVEED, numeric edited move, subroutine was submitted for mutation analysis because it had not been fully tested by conventional means. The program as modified is shown below.

```
      SUBROUTINE MOVEED(SOURCE,SLEN,SDEC,DEST,DLEN,PLEN,PDIG,PDEC,
     * PIC,IER)
      LOGICAL SUPRES, NEGNO
      INTEGER X(5), SUB2, SUB1, IHI, PLDIG, IVAR, I, SCOUNT, DESTHI
      INTEGER CHAR, PDIGLN, SDIG, SARRAY(50), PICST, DDEC
      INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
      CHAR MEMORY(310)
      INTEGER IER
```

```
      CHAR PIC(10)
      INTEGER PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      INPUT OUTPUT MEMORY, IER
      INPUT PIC, PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      SUPRES = .TRUE.                                              87
      DO 5 I=1, PLEN                                               88
 5    SARRAY(I) = '0'                                              89
      PLDIG = PDIG - PDEC                                          90
      SDIG = SLEN - SDEC                                           91
      IF(SDEC .EQ. 0) GOTO 11                                   92 93
      SUB1 = PLDIG                                                 94
      SUB2 = (SOURCE + SDIG) - 1                                   95
      DO 10 I=1, SDEC                                              96
      SUB1 = SUB1 + 1                                              97
      SUB2 = SUB2 + 1                                              98
      IF(MEMORY(SUB2) .EQ. '#') IER = 4                        99 100
 10   SARRAY(SUB1) = MEMORY(SUB2)                                 101
      IF(IER .NE. 0) GOTO 101                                 102 103
 11   IF (SDIG .GE. PLDIG) IHI = PLDIG                            106
      IF(SDIG .LT. PLDIG) IHI = SDIG                          107 108
      SUB1 = PLDIG + 1                                            109
      SUB2 = SOURCE + SDIG                                        110
      DO 15 I=1, IHI                                              111
      SUB1 = SUB1 - 1                                             112
      SUB2 = SUB2 - 1                                             113
      IF(MEMORY(SUB2) .EQ. '#') IER = 4                       114 115
 15   SARRAY(SUB1) = MEMORY(SUB2)                                 116
      IF(IER .NE. 0) GOTO 101                                 117 118
 16   SUB1 = (SOURCE + SLEN) - 1                                  119
      CALL UNPACK(MEMORY(SUB1),X,2)                               120
      NEGNO = X(2) .EQ. '-'                                       121
      SUB1 = DEST                                                 122
      SCOUNT = 0                                                  123
      DO 100 I=1, PLEN                                            124
      SUB1 = DEST + I                                             125
      IF((DEST + I) - 1 .GT. (DLEN + DEST) - 1)) GOTO         126 127
      CHAR = PIC(I)                                               128
      IF(PIC(I) .EQ. '9') SUPRES = .FALSE.                    129 130
      IF(SARRAY(SCOUNT + 1) .NE. '0') SUPRES = .FALSE.        131 132
      IF(CHAR .NE. '-') GOTO 20                               133 134
      MEMORY(SUB1 - 1) = ' '                                      135
      IF(I .EQ. 1 .AND. NEGNO) MEMORY(SUB1 - 1) = '-'         136 137
      IF(I .EQ. 1) GOTO 100                                   138 139
      SCOUNT = SCOUNT + 1                                         140
      IF(.NOT. SUPRES) GOTO 99                                141 142
      IF (NEGNO) MEMORY(SUB1 - 1) = '-'                       143 144
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '    145 146
      GOTO 100                                                    147
 20   IF(CHAR .NE. '+') GOTO 30                               148 149
      IF(I .EQ. 1 .AND. NEGNO) MEMORY(SUB1 - 1) = '-'         150 151
      IF(I .EQ. 1 .AND. .NOT. NEGNO) MEMORY(SUB1 - 1) = '+'   152 153
      IF(I .EQ. 1) GOTO 100                                   154 155
      SCOUNT = SCOUNT + 1                                         156
      IF(.NOT. SUPRES) GOTO 99                                157 158
      IF (NEGNO) MEMORY(SUB1 - 1) = '-'                       159 160
      IF (.NOT. NEGNO) MEMORY(SUB1 - 1) = '+'                 161 162
```

```
      IF(MEMORY(SUB1 - 2) .EQ. '+') MEMORY(SUB1 - 2) = ' '          163 164
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '          165 166
      GOTO 100                                                          167
 30   IF(CHAR .NE. '$') GOTO 40                                     168 169
      IF (I .EQ. 1) MEMORY(SUB1 - 1) = '$'                          170 171
      IF(I .EQ. 1) GOTO 100                                         172 173
      SCOUNT = SCOUNT + 1                                               174
      IF(.NOT. SUPRES) GOTO 99                                      175 176
      MEMORY(SUB1 - 1) = '$'                                            177
      IF(MEMORY(SUB1 - 2) .EQ. '$') MEMORY(SUB1 - 2) = ' '          178 179
      GOTO 100                                                          180
 40   IF(CHAR .NE. '*') GOTO 50                                     181 182
      SCOUNT = SCOUNT + 1                                               183
      IF(.NOT. SUPRES) GOTO 99                                      184 185
      MEMORY(SUB1 - 1) = '*'                                            186
      GOTO 100                                                          187
 50   IF(CHAR .NE. 'Z') GOTO 55                                     188 189
      SCOUNT = SCOUNT + 1                                               190
      IF(.NOT. SUPRES) GOTO 99                                      191 192
      MEMORY(SUB1 - 1) = ' '                                            193
      GOTO 100                                                          194
 55   IF(CHAR .NE. '9') GOTO 60                                     195 196
      SCOUNT = SCOUNT + 1                                               197
      MEMORY(SUB1 - 1) = SARRAY(SCOUNT)                                 198
      GOTO 100                                                          199
 60   IF(CHAR .NE. 'B') GOTO 70                                     200 201
      MEMORY(SUB1 - 1) = ' '                                            202
      GOTO 100                                                          203
 70   IF(CHAR .NE. '/') GOTO 80                                     204 205
      MEMORY(SUB1 - 1) = '/'                                            206
      GOTO 100                                                          207
 80   IF(CHAR .NE. 'V') GOTO 81                                     208 209
      GOTO 100                                                          210
 81   IF(CHAR .NE. '.') GOTO 82                                     211 212
      MEMORY(SUB1 - 1) = '.'                                            213
      GOTO 100                                                          214
 82   IF(CHAR .NE. ',') GOTO 83                                     215 216
      IF(.NOT. SUPRES) MEMORY(SUB1 - 1) = ','                       217 218
      IF(SUPRES) MEMORY(SUB1 - 1) = ' '                             219 220
      GOTO 100                                                          221
 83   IER = 3                                                           222
      GOTO 101                                                          223
 99   MEMORY(SUB1 - 1) = SARRAY(SCOUNT)                                 224
100   CONTINUE                                                          225
101   CONTINUE                                                          226
      RETURN                                                            227
      END
```

The data for this subroutine consisted of the following input and input/output data.

**INPUT DATA**

SOURCE - INTEGER data that contains the starting location in memory for the sending field.

SLEN - INTEGER data that specifies the length of the item in memory.

SDEC - INTEGER specifing the number of digits in the fraction part of a number.

DEST - INTEGER data that contains the starting location in memory for the receiving field.

DLEN - INTEGER data that specifies the length of the receiving data item in memory.

PLEN - INTEGER that specifies the length of the PICTURE specification.

PDIG - INTEGER that gives the number of digits in the PICTURE description.

PDEC - INTEGER specifying the number of digits in the fraction part of the PICTURE.

PIC - CHARACTER array which contains the Cobol PICTURE for the edited move.

**INPUT/OUTPUT DATA**

MEMORY - CHARACTER data that contains the programs memory.

IER - INTEGER used as error indicator.

The numeric edited move takes data from a source field and places it in a receiving field according to what may be called a template or instructions specified in the Cobol PICTURE.

Two errors and redundant conditional statements were found in MOVEED. The first error detected involved a Fortran DO loop where the upperbound on the loop was zero so the DO loop was being executed once when it should not be executed at all. The specific statement is:

```
DO 15 I=1,IHI
```

at line 111 in Figure 5 where IHI has been assigned the value of SDIG (number of digits in the whole part of a number) or PLDIG (number of allowable digits in the whole part of the PICTURE description). The test data that uncovered this error is in Figure 1.

```
TEST CASE NUMBER      9
PARAMETERS ON INPUT
SOURCE = 294
SLEN = 7
SDEC = 7
DEST = 5
DLEN = 8
PLEN = 8
PDIG = 7
PDEC = 2
PIC = "ZZZZ9.99##"
IER = 0
MEMORY = "################################                  00101-      UUUUU
*A      ZZZZZZZZZZ            05             10-             235787                 ZZZ9
*.99                ++++.9                   $$$$$V                    $*****9.99
*           9,999.9              99/99/99                     99B99B99       XXXXXXXX
*XXXXXXXXXXXX       YYYYYYYY3040210200ABCDEELSE2IF2ELSE120301DONE#############
*############################UUUUUAZZZZZZZZZZ                000500001000-01234567##
*#######"
PARAMETERS ON OUTPUT
MEMORY = "#### 1234.56####################                  00101-      UUUUU
*A      ZZZZZZZZZZ            05             10-             235787                 ZZZ9
*.99                ++++.9                   $$$$$V                    $*****9.99
*           9,999.9              99/99/99                     99B99B99       XXXXXXXX
*XXXXXXXXXXXX       YYYYYYYY3040210200ABCDEELSE2IF2ELSE120301DONE#############
*############################UUUUUAZZZZZZZZZZ                000500001000-01234567##
*#######"
IER = 0
```

Figure 1. Test Data Detecting DO Loop Error

The program was corrected and the effected lines for the new program are shown in Figure 2. The new line is the line with the Fortran statement label 11.

```
 11     IF(SDIG .EQ. 0 .OR. PLDIG .EQ. 0) GOTO 16                 104 105
        IHI = PLDIG                                                   106
        IF(SDIG .LT. PLDIG) IHI = SDIG                            107 108
        SUB1 = PLDIG + 1                                              109
        SUB2 = SOURCE + SDIG                                          110
        DO 15 I=1, IHI                                                111
        SUB1 = SUB1 - 1                                               112
        SUB2 = SUB2 - 1                                               113
        IF(MEMORY(SUB2) .EQ. '#') IER = 4                         114 115
 15     SARRAY(SUB1) = MEMORY(SUB2)                                   116
```

Figure 2. Corrected Program

The second error that was uncovered by mutation analysis involved the handling of the PICTURE item 'V' which says not to output a decimal point to the receiving field.

```
TEST CASE NUMBER     1
PARAMETERS ON INPUT
SOURCE = 294
SLEN = 8
SDEC = 4
DEST = 5
DLEN = 7
PLEN = 8
PDIG = 7
PDEC = 3
PIC = "9999V999  "
IER = 0
MEMORY = "##############################                    00101-     UUUUU
*A     ZZZZZZZZZZ          05            10-          235787              ZZZ9
*.99             ++++.9               $$$$$V                $*****9.99
*          9,999.9            99/99/99                 99B99B99     XXXXXXXX
*XXXXXXXXXXXX      YYYYYYYYY3040210200ABCDEELSE2IF2ELSE120301DONE#############
*##############################UUUUUAZZZZZZZZZZ          00050000100#12345678###
*######"
PARAMETERS ON OUTPUT
MEMORY = "####1234567###################                    00101-     UUUUU
*A     ZZZZZZZZZZ          05            10-          235787              ZZZ9
*.99             ++++.9               $$$$$V                $*****9.99
*          9,999.9            99/99/99                 99B99B99     XXXXXXXX
*XXXXXXXXXXXX      YYYYYYYYY3040210200ABCDEELSE2IF2ELSE120301DONE#############
*##############################UUUUUAZZZZZZZZZZ          00050000100#12345678###
*######"
IER = 0
```

Figure 3. Data Detecting PICTURE Clause Error

This error was detected from the data shown in Figure 3. In statement label 80, if a V is the item in the picture, then nothing is done and control goes back to the top of the loop where the next item in the PICTURE description is retrieved. The error occurs because the pointer (variable SUB1) for the next available location in the receiving field is automatically incremented at the beginning of the loop; to correct this error subtract 1 from SUB1 when a V instruction is detected. The original method for calculating the next available location used the Do loop index and the absolute location of the destination field which disregards the statement

SUB1=SUB-1 executed when a 'V' is encountered. This made it mandatory to rewrite the handling of the destination pointer. The new code is given in Appendix D. It has been indicated that some conditional statements were redundant in the original program. These have been rewritten as in Appendix D. Figure 5 contains the program with the 'V' error and with the redundant statements. It can be seen from this listing that several redundant conditional statements have no effect on the result of the program. These redundant statements have been deleted.

Specifically, a redundant conditional statement exists for statement 106 107 where IHI is assigned the value of PLDIG if SDIG is greater than or equal to PLDIG; but, the next statement 108 109 will reassign the value of IHI to SDIG if SDIG is less than PLDIG; it can be seen that the first conditional statement can be changed to the assignment statement IHI=PLDIG because it will be reassigned if the following conditional statement is true.

Another redundant conditional statement is 136 137 where the statement:

```
IF (I .EQ. 1 .AND. NEGNO) MEMORY(SUB1 - 1) = '-'
```

does not need the compound conditional portion I .EQ. 1 because statement 138 139 takes care of that portion of the conditional. This is rewritten as: IF (NEGNO) MEMORY(SUB1 - 1) = '-' which allows the deletion of statement 143 144.

As in the previous conditional statement, the statements 150 151 and 152 153 do not need the portion of the conditional I .EQ. 1 because the statement 154 155 takes care of the condition; also statement 159 160 and statement 161 162 are deleted.

The conditional statement 170 171 is changed to the assignment statement which allows for the deletion of statement 177.

The rewritten MOVEED was tested and the results indicated that the routine was correct. Figure 4 contains the status information for the testing of subroutine MOVEED.

MUTANT ELIMINATION PROFILE FOR MOVEED

| MUTANT TYPE | TOTAL | DEAD | | LIVE | | EQUIV | |
|---|---|---|---|---|---|---|---|
| CONSTANT REPLACEMENT | 151 | 146 | 96.7% | 0 | 0.0% | 5 | 3.3% |
| SCALAR VARIABLE REPLACEME | 2430 | 2413 | 99.3% | 0 | 0.0% | 17 | 0.7% |
| SCALAR FOR CONSTANT REP. | 1121 | 1119 | 99.8% | 0 | 0.0% | 2 | 0.2% |
| CONSTANT FOR SCALAR REP. | 694 | 692 | 99.7% | 0 | 0.0% | 2 | 0.3% |
| SOURCE CONSTANT REPLACEME | 601 | 599 | 99.7% | 0 | 0.0% | 2 | 0.3% |
| ARRAY REF. FOR CONSTANT R | 470 | 470 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| ARRAY REF. FOR SCALAR REP | 1041 | 1030 | 98.9% | 0 | 0.0% | 11 | 1.1% |
| COMPARABLE ARRAY NAME RE | 148 | 148 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| CONSTANT FOR ARRAY REF RE | 105 | 105 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| SCALAR FOR ARRAY REF REP. | 684 | 680 | 99.4% | 0 | 0.0% | 4 | 0.6% |
| ARRAY REF. FOR ARRAY REF. | 251 | 246 | 98.0% | 0 | 0.0% | 5 | 2.0% |
| UNARY OPERATOR INSERTION | 325 | 318 | 97.8% | 0 | 0.0% | 7 | 2.2% |
| ARITHMETIC OPERATOR REPLA | 218 | 218 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| RELATIONAL OPERATOR REPLA | 210 | 191 | 91.0% | 0 | 0.0% | 19 | 9.0% |
| LOGICAL CONNECTOR REPLACE | 5 | 5 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| ABSOLUTE VALUE INSERTION | 399 | 151 | 37.8% | 0 | 0.0% | 248 | 62.2% |
| STATEMENT ANALYSIS | 80 | 80 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| STATEMENT DELETION | 56 | 56 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| RETURN STATEMENT REPLACEM | 128 | 128 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| GOTO STATEMENT REPLACEMEN | 648 | 636 | 98.1% | 0 | 0.0% | 12 | 1.9% |
| DO STATEMENT END REPLACEM | 76 | 72 | 94.7% | 0 | 0.0% | 4 | 5.3% |

MUTANT STATE FOR MOVEED

FOR EXPERIMENT "MOVEED " THIS IS RUN 18

NUMBER OF TEST CASES = 65

NUMBER OF MUTANTS = 9841
NUMBER OF DEAD MUTANTS = 9503 ( 96.6%)
NUMBER OF LIVE MUTANTS = 0 ( 0.0%)
NUMBER OF EQUIV MUTANTS = 338 ( 3.4%)

NUMBER OF MUTANTS WHICH DIED BY NON STANDARD MEANS 4530
NORMALIZED MUTANT RATIO *****%
NUMBER OF MUTATABLE STATEMENTS = 133
GIVING A MUTANTS/STATEMENT RATIO OF 73.99

NUMBER OF DATA REFERENCES = 272
NUMBER OF UNIQUE DATA REFERENCES = 34

ALL MUTANT TYPES HAVE BEEN ENABLED

Figure 4

## Testing Operational Software

The software in these studies was contributed by the U.S. Army Computer Systems Command (Army Institute for Research in Management Information and Computer Science). Both programs are large Cobol modules that had been designed, coded, tested and fielded by the Army. The testers did not have access to the original programmers, but test data was supplied by the Army. The first program was a 2500 line program which was supplied with test data but not documentation or other information to guid the tester. Over 650,000 mutants were generated and run on 3,000 Army test cases. After one week of elapsed testing time, the tester terminated the run when it was determined that the Army supplied test data was of such low quality that less than 10% of the mutants had been eliminated.

The second program is an editor. It consists of 1200 source code lines written in a standard dialect of Cobol. When supplied with a transaction file, the program sorts and edits the input data to generate an error listing with critical and non-critical errors indicated. After all critical errors are corrected and edited, a master file is updated. The updated master file is sorted and a run report is generated.

Minor modifications were required to make the program conform to Level 1 Cobol. Since Level 1 Cobol does not allow multiple data records in a file description, each data record in a such a file was assigned its own file. Since Level 1 Cobol files are specified to be nonrewindable, the program was divided into four sections so that the output of the first section was the input of the second section and so on.

LOW and HIGH values and the current DATE were input by separate files since the CPMS did not supply these values.

Since the purpose of this run was to evaluate the quality of test data supplied by another test organization, the mutation tester did not follow the level-by-level testing strategy suggested in Chapter 2; rather, a<u>ll</u> mutant operators were enabled (see the description of a Level 1 Cobol analyzer in Chapter 2 for a list of Cobol mutant operators). After processing 29 Army test cases, the analyzer returned the following status report.

MUTANT STATUS

| TYPE | TOTAL | LIVE | PCT | EQUIV |
|---|---|---|---|---|
| DECIML | 69 | 48 | 30.43 | 0 |
| OCCURS | 6 | 4 | 33.33 | 0 |
| INSERT | 430 | 100 | 76.74 | 0 |
| FILLSZ | 310 | 45 | 85.48 | 0 |
| ITEMRV | 293 | 77 | 73.72 | 0 |
| FILES | 464 | 0 | 100.00 | 0 |
| DELETE | 545 | 59 | 89.17 | 0 |
| GO PER | 45 | 7 | 84.44 | 0 |
| PER GO | 20 | 3 | 85.00 | 0 |
| IF REV | 75 | 2 | 97.33 | 0 |
| STOP | 541 | 8 | 98.52 | 0 |
| THRU | 365 | 29 | 92.05 | 0 |
| TRAP | 545 | 6 | 98.90 | 0 |
| ARITH | 135 | 17 | 87.41 | 0 |
| ROUND | 45 | 0 | 100.00 | 45 |
| MOVE R | 111 | 5 | 95.50 | 0 |
| LOGIC | 681 | 161 | 76.36 | 0 |
| SUBSFS | 11352 | 947 | 91.66 | 0 |
| SUBCFS | 1004 | 167 | 83.37 | 0 |
| SUBCFS | 1380 | 115 | 91.67 | 0 |
| SUBSFC | 4857 | 457 | 90.59 | 0 |
| C ADJ | 33 | 3 | 90.91 | 0 |
| TOTALS | | | | |
| | 23306 | 2260 | 90.30 | 45 |

This test was augmented by 10 additional cases supplied by the tester and equivalent mutants were removed from the system, resulting in the following mutant status report

---------- 236 MARKED AS EQUIVALENT ----------

MUTANT STATUS

| TYPE | TOTAL | LIVE | PCT | EQUIV |
|---|---|---|---|---|
| DECIML | 69 | 4 | 94.20 | 44 |
| OCCURS | 8 | 2 | 66.67 | 2 |
| INSERT | 430 | 10 | 97.67 | 90 |
| FILLSZ | 310 | 4 | 98.71 | 41 |
| ITEMRV | 293 | 26 | 91.13 | 51 |
| FILES | 464 | 0 | 100.00 | 0 |
| DELETE | 545 | 56 | 89.72 | 3 |
| GO PER | 45 | 6 | 86.67 | 1 |
| PER GO | 20 | 3 | 85.00 | 0 |
| IF REV | 75 | 2 | 97.33 | 0 |
| STOP | 541 | 7 | 98.71 | 1 |
| THRU | 365 | 29 | 92.05 | 0 |
| TRAP | 545 | 3 | 99.45 | 3 |
| ARITH | 135 | 17 | 87.41 | 0 |
| ROUND | 45 | 0 | 100.00 | 45 |
| MOVE R | 111 | 5 | 95.50 | 0 |
| LOGIC | 681 | 161 | 76.36 | 0 |
| SUBSFS | 11352 | 947 | 91.66 | 0 |
| SUBCFC | 1004 | 167 | 83.37 | 0 |
| SUBCFS | 1380 | 115 | 91.67 | 0 |
| SUBSFC | 4857 | 457 | 90.59 | 0 |
| C ADJ | 33 | 3 | 90.91 | 0 |
| TOTALS | | | | |
| | 23306 | 2024 | 91.32 | 281 |

During the analysis of TRAP mutants, a test case was constructed to kill the mutants associated with the report type and the transaction code. The possible values of the type of a report were K,I,W,L,D, and E. The possible transaction values were A,C, and D. The test case constructed consisted of all possible combinations of the report type and the transaction code. The values of other input variables remained the same in each combination.

The interpreter generated a "reference to undefined data at or near line [line number]" error when the program was run on the test case constructed. The statement marked with boldface in the following piece of code was in error.

```
0200-PRINT-ERRORS.
     IF WS-SW2 = 1
     PERFORM 0230-CHECK-FOR-A THRU 0240-EXIT.
     ..........
     ..........
     MOVE STATIONID-2 TO STATIONID-WS-EDIT.
     MOVE INSTALLCODE-02 TO INST-WS-EDIT.
     MOVE TRANSCODE-02 TO TRANSCODE-WS-EDIT.
     ..........
```

The cause of this error was that all elementary data items but one in paragraph 0230-CHECK-FOR-A had been assigned values. The following piece of code shows the paragraph under consideration.

```
0230-CHECK-FOR-A.
     ...............
     ...............
     MOVE WS-STATIONID-WS-K TO STATIONID-WS-EDIT.
     MOVE WS-TRANSCODE-WS-K TO TRANSCODE-WS-EDIT.
     .............
```

There are two ways to correct the error. One solution is to insert the missing statement MOVE WS-INSTALLCODE-WS-K TO INST-WS-EDIT after the line highlighted in boldface. The other solution is to insert the statement MOVE SPACES TO EDITDETAIL-WS after the statement 0200-PRINT-ERRORS. after the statement 0200-PRINT-ERRORS.

## Program A1

```
IDENTIFICATION DIVISION.
PROGRAM-ID. POQAACA.
AUTHOR. CPT R W MOREHEAD.
INSTALLATION. HQS USACSC.
DATE-WRITTEN. OCT 1973.
REMARKS.
    THIS PROGRAM PRINTS OUT A LIST OF CHANGES IN THE ETF.
    ALL ETF CHANGES WERE PROCESSED PRIOR TO THIS PROGRAM.  THE
    OLD ETF AND THE NEW ETF ARE THE INPUTS.  BUT THERE IS NO
    FURTHER PROCESSING OF THE ETF HERE.  THE ONLY OUTPUT IS A
    LISTING OF THE ADDS, CHANGES, AND DELETES.  THIS PROGRAM IS
    FOR HQ USE ONLY AND HAS NO APPLICATION IN THE FIELD.
     *****************
     MODIFIED FOR TESTING UNDER CPMS BY ALLEN ACREE
     JULY, 1979.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. PRIME.
OBJECT-COMPUTER. PRIME.
INPUT-OUTPUT SECT.ON.
FILE-CONTROL.
    SELECT OLD-ETF ASSIGN INPUT4.
    SELECT NEW-ETF ASSIGN INPUT8.
    SELECT PRNTR ASSIGN TO OUTPUT9.
DATA DIVISION.
FILE SECTION.
FD  OLD-ETF
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS OLD-REC.
01  OLD-REC.
    03  FILLER                                  PIC X.
    03  OLD-KEY                                 PIC X(12).
    03  FILLER                                  PIC X(67).
FD  NEW-ETF
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS NEW-REC.
01  NEW-REC.
    03  FILLER                                  PIC X.
    03  NEW-KEY                                 PIC X(12).
    03  FILLER                                  PIC X(67).
FD  PRNTR
    RECORD CONTAINS 40 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRNT-LINE.
01  PRNT-LINE                                   PIC X(40).
WORKING-STORAGE SECTION.
01  PRNT-WORK-AREA.
    03  LINE1                                   PIC X(30).
    03  LINE2                                   PIC X(30).
    03  LINE3                                   PIC X(20).
01  PRNT-OUT-OLD.
    03  WS-LN-1.
        05  FILLER                              PIC X VALUE SPACE.
        05  FILLER                              PIC XXXX VALUE 'O   '.
        05  LN1                                 PIC X(30).
        05  FILLER                              PIC XXX VALUE SPACES.
    03  WS-LN-2.
        05  FILLER                              PIC X VALUE SPACE.
        05  FILLER                              PIC XXXX VALUE 'L   '.
```

```
               05  LN2                                   PIC X(30).
               05  FILLER                                PIC XXX VALUE SPACES.
           03  WS-LN-3.
               05  FILLER                                PIC X VALUE SPACE.
               05  FILLER                                PIC XXXX VALUE 'D   '.
               05  LN3                                   PIC X(20).
               05  FILLER                                PIC XXX  VALUE SPACE.
       01  PRNT-NEW-OUT.
           03  NEW-LN-1.
               05  FILLER                                PIC XXXXX VALUE ' N    '.
               05  N-LN1                                 PIC X(30).
               05  FILLER                                PIC XXX VALUE SPACE.
           03  NEW-LN-2.
               05  FILLER                                PIC XXXXX VALUE ' E    '.
               05  N-LN2                                 PIC X(30).
               05  FILLER                                PIC XXX VALUE SPACES.
           03  NEW-LN-3.
               05  FILLER                                PIC XXXXX VALUE ' W    '.
               05  N-LN3                                 PIC X(20).
               05  FILLER                                PIC XXX VALUE SPACES.
       PROCEDURE DIVISION.
       0100-OPENS.
           OPEN INPUT OLD-ETF NEW-ETF.
           OPEN OUTPUT PRNTR.
       0110-OLD-READ.
           READ OLD-ETF AT END GO TO 0160-OLD-EOF.
       0120-NEW-READ.
           READ NEW-ETF AT END GO TO 0170-NEW-EOF.
       0130-COMPARES.
           IF OLD-KEY = NEW-KEY
               NEXT SENTENCE
           ELSE GO TO 0140-CK-ADD-DEL.
           IF OLD-REC = NEW-REC
               GO TO 0110-OLD-READ.
           MOVE OLD-REC TO PRNT-WORK-AREA.
           PERFORM 0210-OLD-WRT THRU 0210-EXIT.
           MOVE NEW-REC TO PRNT-WORK-AREA.
           PERFORM 0200-NW-WRT THRU 0200-EXIT.
           GO TO 0110-OLD-READ.
       0140-CK-ADD-DEL.
           IF OLD-KEY > NEW-KEY
               MOVE NEW-REC TO PRNT-WORK-AREA
               PERFORM 0200-NW-WRT THRU 0200-EXIT
               GO TO 0120-NEW-READ
           ELSE GO TO 0150-CK-ADD-DEL.
       0150-CK-ADD-DEL.
           MOVE OLD-REC TO PRNT-WORK-AREA.
           PERFORM 0210-OLD-WRT THRU 0210-EXIT.
           READ OLD-ETF AT END
               MOVE NEW-REC TO PRNT-WORK-AREA
               PERFORM 0200-NW-WRT THRU 0200-EXIT
               GO TO 0160-OLD-EOF.
           GO TO 0130-COMPARES.
       0160-OLD-EOF.
           READ NEW-ETF AT END GO TO 0180-EOJ.
           MOVE NEW-REC TO PRNT-WORK-AREA.
           PERFORM 0200-NW-WRT THRU 0200-EXIT.
           GO TO 0160-OLD-EOF.
       0170-NEW-EOF.
           MOVE OLD-REC TO PRNT-WORK-AREA.
           PERFORM 0210-OLD-WRT THRU 0210-EXIT.
           READ OLD-ETF AT END GO TO 0180-EOJ.
           GO TO 0170-NEW-EOF.
       0180-EOJ.
```

```
           CLOSE OLD-ETF NEW-ETF PRNTR.
           STOP RUN.
       0200-NW-WRT.
           MOVE LINE1 TO N-LN1.
           MOVE LINE2 TO N-LN2.
           MOVE LINE3 TO N-LN3.
           WRITE PRNT-LINE FROM NEW-LN-1 AFTER ADVANCING 2.
           WRITE PRNT-LINE FROM NEW-LN-2 AFTER ADVANCING 1.
           WRITE PRNT-LINE FROM NEW-LN-3 AFTER ADVANCING 1.
       0200-EXIT.
           EXIT.
       0210-OLD-WRT.
           MOVE LINE1 TO LN1.
           MOVE LINE2 TO LN2.
           MOVE LINE3 TO LN3.
           WRITE PRNT-LINE FROM WS-LN-1 AFTER ADVANCING 2.
           WRITE PRNT-LINE FROM WS-LN-2 AFTER ADVANCING 1.
           WRITE PRNT-LINE FROM WS-LN-3 AFTER ADVANCING 1.
       0210-EXIT.
           EXIT.
```

## Program A2

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    PROG-1.
AUTHOR.
    JAMES L. BINGHAM.
DATE-WRITTEN.
    APRIL 14, 1979.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. PRIME.
OBJECT-COMPUTER. PRIME.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IN-TRANSACTION ASSIGN TO INPUT0.
    SELECT OUTPUT-PAYMENT ASSIGN TO OUTPUT0.

DATA DIVISION.
FILE SECTION.

FD  IN-TRANSACTION
    RECORD CONTAINS 18 CHARACTERS,
    LABEL RECORDS ARE OMITTED,
    DATA RECORD IS TRANSACTION-RECORD.
01  TRANSACTION-RECORD.
    05 ACCT-NUM                         PIC 9(8).
    05 BILLED-AMT                       PIC 9(5)V99.
    05 PERCENTAGE                       PIC V99.
    05 ACCT-CLASS                       PIC X.

FD  OUTPUT-PAYMENT
    RECORD CONTAINS 55 CHARACTERS,
    LABEL RECORDS ARE OMITTED,
    DATA RECORD IS OUTPUT-RECORD.
01  OUTPUT-RECORD                        PIC X(55).

WORKING-STORAGE SECTION.

01  W-TOTALS-OUTPUT-RECORD.
    05 FILLER                           PIC X(4) VALUE SPACES.
    05 NAME-OF-CLASS                    PIC X(34).
    05 TOTAL-CLASS-PAY                  PIC $$$$$$9.99.
    05 FILLER                           PIC X(4) VALUE SPACES.

01  W-OUTPUT-RECORD.
    05 FILLER                           PIC XXX VALUE SPACES.
    05 W-ACCT-NUM                       PIC 9(8).
    05 FILLER                           PIC XXX VALUE SPACES.
    05 W-BILLED-AMT                     PIC 9(5).99.
    05 FILLER                           PIC XXX VALUE SPACES.
    05 W-PERCENTAGE                     PIC .99.
    05 FILLER                           PIC XXX VALUE SPACES.
    05 W-ACCT-CLASS                     PIC X.
    05 FILLER                           PIC XXX VALUE SPACES.
    05 W-PAYMENT                        PIC $$$$$9.99.

01  TEMPORARY-ITEMS.
    05 TOTAL-A-PAY                      PIC 9(6)V99.
    05 TOTAL-X-PAY                      PIC 9(6)V99.
    05 TOTAL-M-PAY                      PIC 9(6)V99.
    05 TOTAL-T-PAY                      PIC 9(6)V99.
```

```
           05 TOTAL-Z-PAY                         PIC 9(6)V99.
           05 PAY-AMT-A                           PIC 9(5)V99.
           05 PAY-AMT-X                           PIC 9(5)V99.
           05 PAY-AMT-M                           PIC 9(5)V99.
           05 PAY-AMT-T                           PIC 9(5)V99.
           05 PAY-AMT-Z                           PIC 9(5)V99.

       01  ERROR-MESSAGE.
           05 INVALID-DATA-RECORD                  PIC X(50)
              VALUE 'INVALID DATA ON THIS CARD'.

       01  FLAG-VALUE.
           05 MORE-DATA-REMAINS                    PIC X VALUE 'Y'.
      *       88 NO-MORE-DATA-REMAINS                   VALUE 'N'.

       PROCEDURE DIVISION.
       PROCESS-TRANSACTION.
           OPEN INPUT IN-TRANSACTION
                OUTPUT OUTPUT-PAYMENT.
           MOVE ZEROES TO TOTAL-A-PAY, TOTAL-X-PAY, TOTAL-M-PAY,
                          TOTAL-T-PAY, TOTAL-Z-PAY.
           READ IN-TRANSACTION
                AT END MOVE 'N' TO MORE-DATA-REMAINS.
           PERFORM CHECK-DATA UNTIL MORE-DATA-REMAINS = 'N'.
           PERFORM WRITE-OUTPUT-TOTALS.
           CLOSE IN-TRANSACTION
                 OUTPUT-PAYMENT.
           STOP RUN.

       CHECK-DATA.
           IF      ACCT-NUM     IS NUMERIC
              AND BILLED-AMT   IS NUMERIC
              AND PERCENTAGE   IS NUMERIC
              AND  (ACCT-CLASS = 'A' OR
                    ACCT-CLASS = 'X' OR
                    ACCT-CLASS = 'M' OR
                    ACCT-CLASS = 'T' OR
                    ACCT-CLASS = 'Z')
              PERFORM PROCESS-ONE-TRANSACTION
           ELSE
              WRITE OUTPUT-RECORD FROM ERROR-MESSAGE.
           READ IN-TRANSACTION
                AT END MOVE 'N' TO MORE-DATA-REMAINS.

       PROCESS-ONE-TRANSACTION.
           MOVE ACCT-NUM   TO W-ACCT-NUM.
           MOVE BILLED-AMT TO W-BILLED-AMT.
           MOVE PERCENTAGE TO W-PERCENTAGE.
           MOVE ACCT-CLASS TO W-ACCT-CLASS.

           IF ACCT-CLASS = 'A' OR ACCT-CLASS = 'X'
               COMPUTE PERCENTAGE = 1.00 - PERCENTAGE
               IF ACCT-CLASS = 'A'
                   MULTIPLY BILLED-AMT BY PERCENTAGE
                            GIVING PAY-AMT-A ROUNDED
                   ADD PAY-AMT-A TO TOTAL-A-PAY
                   MOVE PAY-AMT-A TO W-PAYMENT
               ELSE
                   MULTIPLY BILLED-AMT BY PERCENTAGE
                            GIVING PAY-AMT-X ROUNDED
                   ADD PAY-AMT-X TO TOTAL-X-PAY
                   MOVE PAY-AMT-X TO W-PAYMENT.

           IF ACCT-CLASS = 'M'
```

```
               MULTIPLY BILLED-AMT BY PERCENTAGE
                         GIVING PAY-AMT-M ROUNDED
               ADD PAY-AMT-M TO TOTAL-M-PAY
               MOVE PAY-AMT-M TO W-PAYMENT.

           IF ACCT-CLASS = 'T'
               MOVE BILLED-AMT TO PAY-AMT-T
               ADD PAY-AMT-T TO TOTAL-T-PAY
               MOVE PAY-AMT-T TO W-PAYMENT.

           IF ACCT-CLASS = 'Z'
               MOVE BILLED-AMT TO PAY-AMT-Z
               ADD PAY-AMT-Z TO TOTAL-Z-PAY
               MOVE PAY-AMT-Z TO W-PAYMENT.

           WRITE OUTPUT-RECORD FROM W-OUTPUT-RECORD.

       WRITE-OUTPUT-TOTALS.
           MOVE TOTAL-A-PAY TO TOTAL-CLASS-PAY.
           MOVE '  TOTAL AMOUNT FOR CLASS A:  ' TO NAME-OF-CLASS.
           WRITE OUTPUT-RECORD FROM W-TOTALS-OUTPUT-RECORD.

           MOVE TOTAL-X-PAY TO TOTAL-CLASS-PAY.
           MOVE '  TOTAL AMOUNT FOR CLASS X:  ' TO NAME-OF-CLASS.
           WRITE OUTPUT-RECORD FROM W-TOTALS-OUTPUT-RECORD.

           MOVE TOTAL-M-PAY TO TOTAL-CLASS-PAY.
           MOVE '  TOTAL AMOUNT FOR CLASS M:  ' TO NAME-OF-CLASS.
           WRITE OUTPUT-RECORD FROM W-TOTALS-OUTPUT-RECORD.

           MOVE TOTAL-T-PAY TO TOTAL-CLASS-PAY.
           MOVE '  TOTAL AMOUNT FOR CLASS T:  ' TO NAME-OF-CLASS.
           WRITE OUTPUT-RECORD FROM W-TOTALS-OUTPUT-RECORD.

           MOVE TOTAL-Z-PAY TO TOTAL-CLASS-PAY.
           MOVE '  TOTAL AMOUNT FOR CLASS Z:  ' TO NAME-OF-CLASS.
           WRITE OUTPUT-RECORD FROM W-TOTALS-OUTPUT-RECORD.
```

## Program A3

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE-4.
REMARKS.  ADAPTED FROM YOURDAN, ET AL. "LEARNING TO PROGRAM
          IN STRUCTURED COBOL."
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  PRIME.
OBJECT-COMPUTER.  PRIME.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT APPLICATION-CARDS-FILE ASSIGN TO INPUTO.
    SELECT PROFILE-LISTING        ASSIGN TO OUTPUTO.
DATA DIVISION.
FILE SECTION.
FD  APPLICATION-CARDS-FILE
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS NAME-ADDRESS-AND-PHONE-IN.
01  NAME-ADDRESS-AND-PHONE-IN.
    05  NAME-IN                               PIC X(20).
    05  ADDRESS-IN                            PIC X(40).
    05  PHONE-IN                              PIC X(11).
    05  FILLER                                PIC X(3).
    05  ACCT-NUM-IN1                          PIC 9(6).
FD  PROFILE-LISTING
    RECORD CONTAINS 132 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINT-LINE-OUT.
01  PRINT-LINE-OUT                            PIC X(132).
WORKING-STORAGE SECTION.
01  COMMON-WS.
    05  CARDS-LEFT                            PIC X(3).
01  CREDIT-INFORMATION-IN.
    05  CARD-TYPE-IN                          PIC X.
    05  ACCT-NUM-IN2                          PIC 9(6).
    05  FILLER                                PIC X.
    05  CREDIT-INFO-IN                        PIC X(22).
    05  FILLER                                PIC X(50).
01  APPLICATION-DATA-WSB1.
    05  NAME-AND-ADDRESS-WS.
        10  NAME-WS                           PIC X(20).
        10  ADDRESS-WS.
            15  STREET-WS                     PIC X(20).
            15  CITY-WS                       PIC X(13).
            15  STATE-WS                      PIC XX.
            15  ZIP-WS                        PIC X(5).
    05  PHONE-WS.
        10  AREA-CODE-WS                      PIC 9(3).
        10  NUMBR-WS                          PIC X(8).
    05  FILLER                                PIC X(3).
    05  ACCT-NUM-WS                           PIC 9(6).
    05  CREDIT-INFO-WS.
        10  SEX-WS                            PIC X.
        10  FILLER                            PIC X.
        10  MARITAL-STATUS-WS                 PIC X.
        10  FILLER                            PIC X.
        10  NUMBER-DEPENS-WS                  PIC X.
```

```
                10  FILLER                          PIC X.
                10  INCOME-HUNDREDS-WS              PIC 9(3).
                10  FILLER                          PIC X.
                10  YEARS-EMPLOYED-WS               PIC 99.
                10  FILLER                          PIC X.
                10  OWN-OR-RENT-WS                  PIC X.
                10  FILLER                          PIC X.
                10  MORTGAGE-OR-RENTAL-WS           PIC 9(3).
                10  FILLER                          PIC X.
                10  OTHER-PAYMENTS-WS               PIC 9(3).
       01  DISCR-INCOME-CALC-FIELDS-WSC8.
           05  ANNUAL-INCOME-WS                     PIC 9(5).
           05  ANNUAL-TAX-WS                        PIC 9(5).
           05  TAX-RATE-WS                          PIC 9V99   VALUE 0.25.
           05  MONTHS-IN-YEAR                       PIC 99     VALUE 12.
           05  MONTHLY-NET-INCOME-WS                PIC 9(4).
           05  MONTHLY-PAYMENTS-WS                  PIC 9(4).
           05  DISCR-INCOME-WS                      PIC S9(3).

       01  LINE-1-WSB3.
           05  FILLER                               PIC X(5) VALUE SPACES.
           05  NAME-L1                              PIC X(20).
           05  FILLER                               PIC X(11)
                   VALUE  '    PHONE ('.
           05  AREA-CODE-L1                         PIC 9(3).
           05  FILLER                               PIC XX  VALUE ') '.
           05  NUMBR-L1                             PIC X(8).
           05  FILLER                               PIC X(3) VALUE SPACES.
           05  SEX-L1                               PIC X(6).
           05  FILLER                               PIC X(9) VALUE SPACES.
           05  FILLER                               PIC X(14)
                    VALUE 'INCOME       $'.
           05  INCOME-HUNDREDS-L1                   PIC 9(3).
           05  FILLER                               PIC X(28)
                    VALUE '00 PER YEAR; IN THIS EMPLOY '.
           05  YEARS-EMPLOYED-L1.
               10  YEARS-L1                         PIC XX.
               10  DESCN-L1                         PIC X(16).
       01  LINE-2-WSB3.
           05  FILLER                               PIC X(5) VALUE SPACES.
           05  STREET-L2                            PIC X(20).
           05  FILLER                               PIC X(27) VALUE SPACES.
           05  MARITAL-STATUS-L2                    PIC X(8).
           05  FILLER                               PIC X(7) VALUE SPACES.
           05  OUTGO-DESCN                          PIC X(16).
           05  MORTGAGE-OR-RENTAL-L2                PIC 9(3).
           05  FILLER                               PIC X(11)
                 VALUE ' PER MTH   '.
           05  FILLER                               PIC X(22)
                 VALUE 'DISCRETIONARY INCOME $'.
           05  DISCR-INCOME-L2                      PIC 9(3).
           05  FILLER                               PIC X(9)
                 VALUE ' PER MTH '.
       01  LINE-3-WSB3.
           05  FILLER                               PIC X(5) VALUE SPACES.
           05  CITY-L3                              PIC X(13).
           05  FILLER                               PIC X VALUE SPACE.
           05  STATE-L3                             PIC XX.
           05  FILLER                               PIC X VALUE SPACE.
           05  ZIP-L3                               PIC X(5).
           05  FILLER                               PIC X(7) VALUE '  A/C: '.
           05  ACCT-NUM-L3                          PIC 9(6).
           05  FILLER                               PIC X(12) VALUE SPACES.
           05  NUMBER-DEPENS-L3                     PIC 9.
```

```
        05  FILLER                          PIC X(14)
                VALUE ' DEPENDENTS    '.
        05  FILLER                          PIC X(16)
                VALUE 'OTHER PAYMENTS $'.
        05  OTHER-PAYMENTS-L3               PIC 9(3).

 PROCEDURE DIVISION.
 A0-MAIN-BODY.
        PERFORM A1-INITIALIZATION.
        PERFORM A2-PRINT-PROFILES
          UNTIL CARDS-LEFT = 'NO '.
        PERFORM A3-END-OF-JOB.
        STOP RUN.

 A1-INITIALIZATION.
        OPEN INPUT     APPLICATION-CARDS-FILE
             OUTPUT    PROFILE-LISTING.
*** USELESS INITIALIZATIONS HAVE BEEN COMMENTED OUT
***     MOVE ZEROES TO ANNUAL-INCOME-WS.
***     MOVE ZEROES TO ANNUAL-TAX-WS.
***     MOVE ZEROES TO MONTHLY-NET-INCOME-WS.
***     MOVE ZEROES TO MONTHLY-PAYMENTS-WS.
***     MOVE ZEROES TO DISCR-INCOME-WS.
        MOVE 'YES' TO CARDS-LEFT.
        READ APPLICATION-CARDS-FILE
            AT END MOVE 'NO ' TO CARDS-LEFT.
*  THE FIRST CARD OF A PAIR IS NOW IN THE BUFFER.

 A2-PRINT-PROFILES.
        PERFORM B1-GET-A-PAIR-OF-CARDS-INTO-WS.
        PERFORM B2-CALC-DISCRETNRY-INCOME.
        PERFORM B3-ASSEMBLE-PRINT-LINES.
        PERFORM B4-WRITE-PROFILE.

 A3-END-OF-JOB.
        CLOSE APPLICATION-CARDS-FILE
              PROFILE-LISTING.

 B1-GET-A-PAIR-OF-CARDS-INTO-WS.
        MOVE NAME-IN TO NAME-WS.
        MOVE ADDRESS-IN TO ADDRESS-WS.
        MOVE PHONE-IN TO PHONE-WS.
        MOVE ACCT-NUM-IN1 TO ACCT-NUM-WS.
        READ APPLICATION-CARDS-FILE INTO CREDIT-INFORMATION-IN
***         AT END MOVE 'NO ' TO CARDS-LEFT.
            AT END MOVE '    *** MISSING SECOND CARD OF PAIR ***'
                        TO PRINT-LINE-OUT
                   WRITE PRINT-LINE-OUT AFTER ADVANCING 2 LINES
                   PERFORM A3-END-OF-JOB
                   STOP RUN.
* THE SECOND CARD OF THE PAIR IS NOW IN THE BUFFER.
        MOVE CREDIT-INFO-IN TO CREDIT-INFO-WS
        READ APPLICATION-CARDS-FILE
            AT END MOVE 'NO ' TO CARDS-LEFT.
* THE FIRST CARD OF THE NEXT PAIR IS NOW IN THE BUFFER.

 B2-CALC-DISCRETNRY-INCOME.
        COMPUTE ANNUAL-INCOME-WS = INCOME-HUNDREDS-WS * 100.
        COMPUTE ANNUAL-TAX-WS    = ANNUAL-INCOME-WS * TAX-RATE-WS.
        COMPUTE MONTHLY-NET-INCOME-WS ROUNDED
            = (ANNUAL-INCOME-WS - ANNUAL-TAX-WS) / MONTHS-IN-YEAR.
        COMPUTE MONTHLY-PAYMENTS-WS = MORTGAGE-OR-RENTAL-WS
                               + OTHER-PAYMENTS-WS.
        COMPUTE DISCR-INCOME-WS = MONTHLY-NET-INCOME-WS
```

```
                                   - MONTHLY-PAYMENTS-WS
             ON SIZE ERROR MOVE 999 TO DISCR-INCOME-WS.
      *   DISCRETIONARY INCOMES OVER $999 PER MONTH ARE SET AT $999.

       B3-ASSEMBLE-PRINT-LINES.
           MOVE NAME-WS TO NAME-L1.
           MOVE STREET-WS TO STREET-L2.
           MOVE CITY-WS TO CITY-L3.
           MOVE STATE-WS TO STATE-L3.
           MOVE ZIP-WS TO ZIP-L3.
           MOVE AREA-CODE-WS TO AREA-CODE-L1.
           MOVE NUMBR-WS TO NUMBR-L1.
           MOVE ACCT-NUM-WS TO ACCT-NUM-L3.
           IF SEX-WS = 'M' MOVE 'MALE  ' TO SEX-L1.
           IF SEX-WS = 'F' MOVE 'FEMALE' TO SEX-L1.
           IF MARITAL-STATUS-WS = 'S' MOVE 'SINGLE  '
                   TO MARITAL-STATUS-L2.
           IF MARITAL-STATUS-WS = 'M' MOVE 'MARRIED '
                   TO MARITAL-STATUS-L2.
           IF MARITAL-STATUS-WS = 'D' MOVE 'DIVORCED'
                   TO MARITAL-STATUS-L2.
           IF MARITAL-STATUS-WS = 'W' MOVE 'WIDOWED '
                   TO MARITAL-STATUS-L2.
           MOVE NUMBER-DEPENS-WS TO NUMBER-DEPENS-L3.
           MOVE INCOME-HUNDREDS-WS TO INCOME-HUNDREDS-L1.
           IF  YEARS-EMPLOYED-WS IS EQUAL TO 0
               MOVE 'LESS THAN 1 YEAR' TO YEARS-EMPLOYED-L1
           ELSE
               MOVE YEARS-EMPLOYED-WS TO YEARS-L1
               MOVE ' YEARS        ' TO DESCN-L1.
           IF OWN-OR-RENT-WS = 'O' MOVE 'MORTGAGE:        $'
                   TO OUTGO-DESCN.
           IF OWN-OR-RENT-WS = 'R' MOVE 'RENTAL:          $'
                   TO OUTGO-DESCN.
           MOVE MORTGAGE-OR-RENTAL-WS TO MORTGAGE-OR-RENTAL-L2.
           MOVE OTHER-PAYMENTS-WS TO OTHER-PAYMENTS-L3.
           MOVE DISCR-INCOME-WS TO DISCR-INCOME-L2.

       B4-WRITE-PROFILE.
      ***  MOVE SPACES TO PRINT-LINE-OUT.
           WRITE PRINT-LINE-OUT FROM LINE-1-WSB3
                   AFTER ADVANCING 4 LINES.
      ***  MOVE SPACES TO PRINT-LINE-OUT.
           WRITE PRINT-LINE-OUT FROM LINE-2-WSB3
                   AFTER ADVANCING 1 LINES.
      ***  MOVE SPACES TO PRINT-LINE-OUT.
           WRITE PRINT-LINE-OUT FROM LINE-3-WSB3
                   AFTER ADVANCING 1 LINES.
```

## Program A4

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SRMFREP.
AUTHOR. R A OVERBEEK.
REMARKS. THIS PROGRAM IS USED TO PRODUCE THE STATUS REPORTS
         BY DEPARTMENT, FOR ALL OF THE STUDENTS RECORDED IN
         THE SRMF.

         ADAPTED TO THE COBCL MUTATION SYSTEM BY ALLEN ACREE.

               ERRORS DISCOVERED:

               (1)  ERRORS IN THE INPUT FILE SETUP, CHECKED FOR
               IN THE PROGRAM, CAUSE REFERENCES TO UNDEFINED
               DATA, PARTICULARLY LINE-COUNT.  CORRECTED WITH
               A VALUE CLAUSE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. CMS.
OBJECT-COMPUTER. CMS.
SPECIAL-NAMES. C01 IS TOP-OF-PAGE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER ASSIGN TO INPUTO.
    SELECT PRINT-FILE ASSIGN TO OUTPUTO.

DATA DIVISION.
FILE SECTION.
FD  MASTER
    RECORD CONTAINS 141 CHARACTERS,
    LABEL RECORDS ARE STANDARD,
    DATA RECORD IS ITEM.
01  ITEM.
    02  SOC-SEC-IN.
        03  SOC-SEC-IN-1                PIC X(3).
        03  SOC-SEC-IN-2                PIC X(2).
        03  SOC-SEC-IN-3                PIC X(4).
    02  NAME-IN                         PIC X(5).
    02  ADDR-IN-1                       PIC X(5).
    02  ADDR-IN-2                       PIC X(5).
    02  MAJOR-IN                        PIC X(4).
    02  STATUS-IN                       PIC X(1).
    02  NO-COURSES                      PIC 99.
    02  COURSE-ENTRY OCCURS 11 TIMES.
        03  DEPT-OFF                    PIC X(2).
        03  COURSE-NO                   PIC X(2).
        03  CREDITS                     PIC 99.
        03  SEMESTER                    PIC X(1).
        03  YEAR                        PIC X(2).
        03  GRADE                       PIC X(1).
FD  PRINT-FILE
    RECORD CONTAINS 89 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINT-BUFF.
01  PRINT-BUFF                          PIC X(89).

WORKING-STORAGE SECTION.
77  END-ALL                             PIC 99.
77  END-MARKER                          PIC 99.
77  P-INDEX                             PIC 9.
77  POINTS                              PIC 999.
77  CR-HRS                              PIC 999.
```

```
       77  INCR                                 PIC 99.
       77  C-INDEX                              PIC 99.
       77  PAGE-NO                              PIC 999 VALUE IS 1.
       77  LINE-COUNT                           PIC 99  VALUE ZERO.
       77  SAVE-KEY                             PIC X(4).
       77  TOT-NO-RECORDS                       PIC 9999999 VALUE IS 0.
       77  SUB-TOT-NO                           PIC 9999999.

       01  HEADER.
           02  FILLER                           PIC X(14).
           02  COLLEGE                          PIC X(30).
           02  DATE-IN                          PIC X(8).
       01  TRAILER.
           02  FILLER                           PIC X(49).
           02  NO-RECORDS                       PIC 9999999.
       01  PRINT-LINE.
           02  FILLER                           PIC X(1).
           02  SOC-SEC-OUT.
               03  SOC-SEC-O1                   PIC X(3).
               03  SOC-SEC-F1                   PIC X(1).
               03  SOC-SEC-O2                   PIC X(2).
               03  SOC-SEC-F2                   PIC X(1).
               03  SOC-SEC-O3                   PIC X(4).
           02  FILLER                           PIC X(2).
           02  NAME-ADDR                        PIC X(5).
           02  FILLER                           PIC X(1).
           02  MAJOR-O                          PIC X(4).
           02  FILLER                           PIC X(1).
           02  STATUS-O                         PIC X(1).
           02  FILLER                           PIC X(1).
           02  GPA                              PIC 9.99.
           02  FILLER                           PIC X(2).
           02  COURSE-O    OCCURS 3 TIMES.
               03  C-DEPT                       PIC X(2).
               03  FILLER                       PIC X(1).
               03  C-NO                         PIC X(2).
               03  FILLER                       PIC X(1).
               03  CREDITS-O                    PIC Z9.
               03  FILLER                       PIC X(1).
               03  SEMESTER-O                   PIC X(1).
               03  DASH-O                       PIC X(1).
               03  YEAR-O                       PIC X(2).
               03  FILLER                       PIC X(2).
               03  GRADE-O                      PIC X(1).
               03  FILLER                       PIC X(2).
           02  FILLER                           PIC X(2).
       01  PAGE-HEADER.
           02  FILLER                           PIC X(4)   VALUE SPACES.
           02  DATE-O                           PIC X(8).
           02  FILLER                           PIC X(17)  VALUE SPACES.
           02  COLL-O                           PIC X(30).
           02  FILLER                           PIC X(17)  VALUE SPACES.
           02  FILLER                           PIC X(5)   VALUE IS 'PAGE'.
           02  PAGE-O                           PIC ZZ9.
           02  FILLER                           PIC X(5)   VALUE SPACES.
       01  COL-HDR-1.
           02  FILLER                           PIC X(20)
                   VALUE ' SOC SEC      N & A '.
           02  FILLER                           PIC X(10) VALUE 'MAJ ST GPA'.
           02  FILLER                           PIC X(9) VALUE SPACES.
           02  FILLER                           PIC X(6) VALUE 'COURSE'.
           02  FILLER                           PIC X(12) VALUE SPACES.
           02  FILLER                           PIC X(6) VALUE 'COURSE'.
           02  FILLER                           PIC X(12) VALUE SPACES.
```

```
           02  FILLER                         PIC X(6) VALUE 'COURSE'.
           02  FILLER                         PIC X(8) VALUE SPACES.
       01  COL-HDR-2.
           02  FILLER                         PIC X(33) VALUE SPACES.
           02  FILLER                         PIC X(18)
                   VALUE ' NMBR CR S-YR  GR '.
           02  FILLER                         PIC X(18)
                   VALUE ' NMBR CR S-YR  GR '.
           02  FILLER                         PIC X(20)
                   VALUE ' NMBR CR S-YR  GR   '.
       01  SUB-TOT-LINE.
           02  FILLER                         PIC X(4)  VALUE SPACES.
           02  FILLER                         PIC X(8)
                   VALUE IS 'TOTAL = '.
           02  SUB-TOT                        PIC ZZZZZZ9.
           02  FILLER                         PIC X(70)  VALUE SPACES.
       PROCEDURE DIVISION.
      * MAIN-PROGRAM SECTION.
       START.
           OPEN INPUT MASTER OUTPUT PRINT-FILE.
           READ MASTER INTO HEADER AT END GO TO EOF.
           IF SOC-SEC-IN IS = SPACES GO TO GOT-HEADER.
           MOVE ' NO HEADER FOUND ON THE MASTER FILE ***' TO PRINT-LINE.
           PERFORM PRINT2-ROUTINE THRU PRINT2-EXIT.
           GO TO CLOSE-FILES.
       GOT-HEADER.
           MOVE COLLEGE TO COLL-O.
           MOVE DATE-IN TO DATE-O.
           READ MASTER AT END GO TO EOF.
           IF SOC-SEC-IN IS NOT = '999999999' GO TO SAVE-DEPT-NAME.
           MOVE ' NO ITEM RECORDS IN MASTER FILE ***' TO PRINT-LINE.
           PERFORM PRINT2-ROUTINE THRU PRINT2-EXIT.
           GO TO CLOSE-FILES.
       SAVE-DEPT-NAME.
           MOVE MAJOR-IN TO SAVE-KEY.
      * NAME OF DEPARTMENT IS SUBTOTAL KEY.  BREAK OCCURS WHENEVER
      * FIELD IS DIFFERENT ON TWO CONSECUTIVE RECORDS.
           MOVE 0 TO SUB-TOT-NO.
           MOVE 1 TO PAGE-NO.
      * PAGE-NO IS RESET TO 1 FOR EACH DEPARTMENT REPORT.
           MOVE 16 TO LINE-COUNT.
           MOVE SPACES TO PRINT-LINE.

       ITEM-LOOP.
           PERFORM ITEM-ROUTINE THRU ITEM-EXIT.
           ADD 1 TO SUB-TOT-NO.
           READ MASTER INTO TRAILER AT END GO TO EOF.
           IF MAJOR-IN IS = SAVE-KEY GO TO ITEM-LOOP.

       DO-SUB-TOTALS.
           MOVE SUB-TOT-NO TO SUB-TOT.
           WRITE PRINT-BUFF FROM SUB-TOT-LINE AFTER ADVANCING 2 LINES.
           ADD SUB-TOT-NO TO TOT-NO-RECORDS.
           IF SOC-SEC-IN IS NOT = '999999999' GO TO SAVE-DEPT-NAME.
           MOVE TOT-NO-RECORDS TO SUB-TOT.
           WRITE PRINT-BUFF FROM SUB-TOT-LINE
                   AFTER ADVANCING TOP-OF-PAGE.
           IF NO-RECORDS IS = TOT-NO-RECORDS GO TO CLOSE-FILES.
           MOVE ' *** MASTER TRAILER VERIFICATION HAS FAILED ***'
                   TO PRINT-LINE.
           PERFORM PRINT2-ROUTINE THRU PRINT2-EXIT.
       CLOSE-FILES.
           CLOSE MASTER PRINT-FILE.
           STOP RUN.
```

```
 EOF.
     MOVE ' EOF ON MASTER FILE ****' TO PRINT-LINE.
     PERFORM PRINT2-ROUTINE THRU PRINT2-EXIT.
     GO TO CLOSE-FILES.

* SUB-ROUTINE SECTION.

 PRINT1-ROUTINE.
     IF LINE-COUNT IS < 16 GO TO NORMAL-PRINT.
     PERFORM HEADER-ROUTINE THRU HEADER-EXIT.
     WRITE PRINT-BUFF FROM PRINT-LINE AFTER ADVANCING 2 LINES.
     ADD 2 TO LINE-COUNT.
     GO TO COMMON-POINT.
 NORMAL-PRINT.
     WRITE PRINT-BUFF FROM PRINT-LINE AFTER ADVANCING 1 LINES.
     ADD 1 TO LINE-COUNT.
 COMMON-POINT.
     MOVE SPACES TO PRINT-LINE.
 PRINT1-EXIT.  EXIT.

 PRINT2-ROUTINE.
     IF LINE-COUNT IS > 14
         PERFORM HEADER-ROUTINE THRU HEADER-EXIT.
     WRITE PRINT-BUFF FROM PRINT-LINE AFTER ADVANCING 2 LINES.
     ADD 2 TO LINE-COUNT.
     MOVE SPACES TO PRINT-LINE.
 PRINT2-EXIT.  EXIT.

 HEADER-ROUTINE.
     MOVE PAGE-NO TO PAGE-O.
     WRITE PRINT-BUFF FROM PAGE-HEADER
         AFTER ADVANCING TOP-OF-PAGE.
     ADD 1 TO PAGE-NO.
     WRITE PRINT-BUFF FROM COL-HDR-1 AFTER ADVANCING 2 LINES.
     WRITE PRINT-BUFF FROM COL-HDR-2 AFTER ADVANCING 1 LINES.
     MOVE 0 TO LINE-COUNT.
 HEADER-EXIT.  EXIT.

 ITEM-ROUTINE.
     MOVE SOC-SEC-IN-1 TO SOC-SEC-O1.
     MOVE SOC-SEC-IN-2 TO SOC-SEC-O2.
     MOVE SOC-SEC-IN-3 TO SOC-SEC-O3.
     MOVE '-' TO SOC-SEC-F1.
     MOVE '-' TO SOC-SEC-F2.
     MOVE NAME-IN TO NAME-ADDR.
     MOVE MAJOR-IN TO MAJOR-O.
     MOVE STATUS-IN TO STATUS-O
*  CALCULATE THE GPA.
     MOVE 0 TO POINTS.
     MOVE 0 TO CR-HRS.
     PERFORM GPA-ACCUM THRU GPA-EXIT VARYING C-INDEX
         FROM 1 BY 1 UNTIL C-INDEX IS > NO-COURSES.
     IF CR-HRS IS = 0 GO TO NO-GPA.
     DIVIDE POINTS BY CR-HRS GIVING GPA ROUNDED.
*        IN THE FOLLOWING THESE INDICES ARE USED:
*        END-ALL: THE INDEX OF THE FIRST UNUSED COURSE
*                 ENTRY; THIS MARKS THE END OF THE COURSES
*                 TO PRINT;
*        END-MARKER: WHEN FILL-LINE IS CALLED END-MARKER
*                 POINTS AT THE FIRST COURSE ENTRY PAST THE
*                 LAST ENTRY TO BE PUT INTO THE LINE;
*        C-INDEX: WHEN FILL-LINE IS CALLED C-INDEX POINTS
*                 AT THE FIRST COURSE ENTRY WHICH GETS
*                 PUT INTO THE PRINT-LINE;  THUS, IF C-INDEX
```

```
*                      IS EQUAL TO END-MARKER, NO COURSE ENTRIES
*                      GET PUT INTO THE PRINT LINE;
*             P-INDEX: INDEXES THE SPOT IN THE PRINT-LINE
*                      WHERE THE ENTRY POINTED TO BY C-INDEX
*                      IS TO BE MOVED;  THUS, ITS RANGE IS 1 TO 3.

 NO-GPA.
     MOVE 1 TO C-INDEX.
     ADD 1 NO-COURSES GIVING END-ALL.
     MOVE 4 TO END-MARKER.
     IF END-ALL IS < END-MARKER MOVE END-ALL TO END-MARKER.
     PERFORM FILL-LINE THRU FILL-EXIT.
     PERFORM PRINT2-ROUTINE THRU PRINT2-EXIT.
     MOVE ADDR-IN-1 TO NAME-ADDR.
     MOVE 7 TO END-MARKER.
     IF END-ALL IS < END-MARKER MOVE END-ALL TO END-MARKER.
     PERFORM FILL-LINE THRU FILL-EXIT.
     PERFORM PRINT1-ROUTINE THRU PRINT1-EXIT.
     MOVE ADDR-IN-2 TO NAME-ADDR.
     MOVE 10 TO END-MARKER.
 COURSE-LOOP.
     IF END-ALL IS < END-MARKER MOVE END-ALL TO END-MARKER.
     PERFORM FILL-LINE THRU FILL-EXIT.
     PERFORM PRINT1-ROUTINE THRU PRINT1-EXIT.
     IF C-INDEX = END-ALL GO TO ITEM-EXIT.
     ADD 3 C-INDEX GIVING END-MARKER.
     GO TO COURSE-LOOP.
 ITEM-EXIT.  EXIT.
 FILL-LINE.
     MOVE 1 TO P-INDEX.
 CHECK-END.
     IF C-INDEX IS = END-MARKER GO TO FILL-EXIT.
     MOVE DEPT-OFF (C-INDEX) TO C-DEPT (P-INDEX).
     MOVE COURSE-NO (C-INDEX) TO C-NO (P-INDEX).
     MOVE CREDITS (C-INDEX) TO CREDITS-O (P-INDEX).
     MOVE SEMESTER (C-INDEX) TO SEMESTER-O (P-INDEX).
     MOVE '-' TO DASH-O (P-INDEX).
     MOVE YEAR (C-INDEX) TO YEAR-O (P-INDEX).
     MOVE GRADE (C-INDEX) TO GRADE-O (P-INDEX).
     ADD 1 TO C-INDEX.
     ADD 1 TO P-INDEX.
     GO TO CHECK-END.
 FILL-EXIT.  EXIT.

 GPA-ACCUM.
     IF GRADE (C-INDEX) IS NOT = 'A' GO TO NOTA.
     MULTIPLY CREDITS (C-INDEX) BY 4 GIVING INCR.
     GO TO COMMON-ADD.
 NOTA.
     IF GRADE (C-INDEX) IS NOT = 'B' GO TO NOTB.
     MULTIPLY CREDITS (C-INDEX) BY 3 GIVING INCR.
     GO TO COMMON-ADD.
 NOTB.
     IF GRADE (C-INDEX) IS NOT = 'C' GO TO NOTC.
     MULTIPLY CREDITS (C-INDEX) BY 2 GIVING INCR.
     GO TO COMMON-ADD.
 NOTC.
     IF GRADE (C-INDEX) IS NOT = 'D' GO TO NOTD.
     MULTIPLY CREDITS (C-INDEX) BY 1 GIVING INCR.
     GO TO COMMON-ADD.
 NOTD.
     IF GRADE (C-INDEX) IS NOT = 'F' GO TO GPA-EXIT.
     MOVE 0 TO INCR.
 COMMON-ADD.
```

```
           ADD INCR TO POINTS.
           ADD CREDITS (C-INDEX) TO CR-HRS.
       GPA-EXIT.  EXIT.
```

**Program A5**

```
IDENTIFICATION DIVISION.
*
*     REPORT CONTAINS THE INPUT DATA ALONG WITH THE
*     CURRENT COMMISSION FOR EACH SALESMAN. AT THE
*     END OF THIS SINGLE SPACED REPORT THE FOLLOWING
*     TOTALS ARE PRINTED: YEAR TO DATE SALES, CUR-
*     RENT SALES, CURRENT COMMISSION.
*
*     CURRENT COMMISSION IS CALCULATED AS FOLLOWS:
*       CURRENT-COMMISSION = CURRENT-SALES *
*         ( COMMISSION-RATE + VOLUME-BONUS + DEPARTMENT-BONUS )
*
*     WITH DEPARTMENT BONUS DETERMINED AS FOLLOWS:
*         DEPT       BONUS
*          01         0.1%
*          02         0.1%
*          04         0.7%
*          05         0.6%
*          06         0.4%
*          07         0.6%
*          09         0.4%
*         OTHER       0.0%
*
*     WITH VOLUME BONUS DETERMINED AS FOLLOWS:
*         AVERAGE MONTHLY SALES      BONUS
*           UNDER $500                0.0%
*           $500 TO $999.99           0.3%
*           $1000 TO $1999.99         0.4%
*           OVER $2000                0.6%
*
*     WITH AVERAGE MONTHS SALES DETERMINED AS FOLLOWS:
*       AVERAGE-MONTHLY-SALES =
*         ( YEAR-TO-DATE-SALES + CURRENT-SALES ) / MONTHS-EMPLOYED

PROGRAM-ID.  COMMISSION-REPORT.

AUTHOR.
    DANIEL CASTAGNO,ICS 3400,STUDENT NUMBER 654,PROGRAM 1.

REMARKS.  SLIGHTLY MODIFIED FOR CMS.1 BY A.ACREE.
          MUTATION TESTING UNCOVERED THE FOLLOWING ERRORS AND
          INEFFICIENCIES:
          (1) REPORT HEADER WITH PAGE ADVANCE WAS NOT PRINTED
          AFTER FULL-PAGE CONDITION RAISED BY INVALID DATA RECORD
          EXTRA PERFORM INSERTED.
          (2) DATA ITEMS DEFINED AND NEVER USED -- DELETED.
          (3) MOVE STATEMENT REPEATED -- SECOND VERSION DELETED.
          (4) TWO USELESS INITIALIZATIONS DELETED.


ENVIRONMENT DIVISION.

CONFIGURATION SECTION.
SOURCE-COMPUTER.
    CYBER-74.
OBJECT-COMPUTER.
    CYBER-74.
SPECIAL-NAMES.
    C01 IS TO-TOP-OF-PAGE.

INPUT-OUTPUT SECTION.
```

```
       FILE-CONTROL.
           SELECT CARD-FILE ASSIGN TO INPUTO.
           SELECT PRINT-FILE ASSIGN TO OUTPUTO.

       DATA DIVISION.

       FILE SECTION.

       FD  CARD-FILE
           RECORD CONTAINS 80 CHARACTERS,
           LABEL RECORDS ARE OMITTED,
           DATA RECORD IS CARD-RECORD.

       01  CARD-RECORD.
           02  I-CARD-DATA.
               03  I-STORE-NUMBER          PIC 99.
               03  I-DEPARTMENT            PIC XX.
               03  I-SALESMAN-NUMBER       PIC 999.
               03  I-SALESMAN-NAME         PIC X(20).
               03  I-YEAR-TO-DATE-SALES    PIC 9(5)V99.
               03  I-CURRENT-SALES         PIC 9(5)V99.
               03  I-COMMISSION-RATE       PIC V99.
               03  I-MONTHS-EMPLOYED       PIC 99.
           02  FILLER                      PIC X(35) .

       FD  PRINT-FILE
           RECORD CONTAINS 132 CHARACTERS,
           LABEL RECORDS ARE OMITTED,
           DATA RECORD IS LINE-RECORD.

       01  LINE-RECORD                     PIC X(132).


       WORKING-STORAGE SECTION.

       77  W-DEPARTMENT-BONUS              PIC V999.
       77  W-VOLUME-BONUS                  PIC V999.
       77  W-DEPARTMENT                    PIC XX.
       77  W-STORE-NUMBER                  PIC 99.
       77  W-SALESMAN-NUMBER               PIC 999.
       77  W-YEAR-TO-DATE-SALES            PIC 9(5)V99.
       77  W-CURRENT-SALES                 PIC 9(5)V99.
       77  W-COMMISSION-RATE               PIC V99.
       77  W-MONTHS-EMPLOYED               PIC 99.
       77  W-CURRENT-COMMISSION            PIC 9(4)V99.
       77  W-TOTAL-YEAR-TO-DATE-SALES      PIC 9(9)V99
           VALUE 0.
       77  W-TOTAL-CURRENT-SALES           PIC 9(8)V99
           VALUE 0.
       77  W-TOTAL-CURRENT-COMMISSION      PIC 9(7)V99
           VALUE 0.
       77  W-AVERAGE-MONTHLY-SALES         PIC 9(7)V99
           VALUE 0.


      *01  KEY-TO-RECORDS.
      *    02  SALESMAN-NUM                PIC 999.

       01  FLAGS.
           02  VALID-DATA-FLAG             PIC  XXX
               VALUE 'YES'.
           02  MORE-DATA-REMAINS-FLAG      PIC  XXX
               VALUE 'YES'.
```

```
01  CONSTANTS.
    02  DEPT.
        03  DEPT-1-OR-2                PIC V999
            VALUE 0.001.
        03  DEPT-6-OR-9                PIC V999
            VALUE 0.004.
        03  DEPT-5-OR-7                PIC V999
            VALUE 0.006.
        03  DEPT-4                     PIC V999
            VALUE C.007.
        03  DEPT-OTHER                 PIC V999
            VALUE 0.000.
    02 VOLUMN.
        03  LEVEL-1                    PIC V999
            VALUE C.
        03  LEVEL-2                    PIC V999
            VALUE 0.003.
        03  LEVEL-3                    PIC V999
            VALUE 0.004.
        03  LEVEL-4                    PIC V999
            VALUE 0.006.

01  COUNTERS.
    02  LINE-COUNT                     PIC 99
        VALUE 0.

01  FINAL-TOTAL-LINE.
    02  FILLER                         PIC X(10)
        VALUE '     TOTAL'.
    02  FILLER                         PIC X(51)
        VALUE SPACES.
    02  O-TOTAL-YEAR-TO-DATE-SALES PIC Z(9).99.
    02  FILLER                         PIC XXX
        VALUE SPACES.
    02  O-TOTAL-CURRENT-SALES          PIC Z(8).99.
    02  FILLER                         PIC X(15)
        VALUE SPACES.
    02  O-TOTAL-CURRENT-COMMISSION PIC Z(7).99.
    02  FILLER                         PIC X(20)
        VALUE SPACES.

01  REPORT-LINE-1.
    02  FILLER                         PIC X(61)
        VALUE SPACES.
    02  FILLER                         PIC X(10)
        VALUE 'COMMISSION'.
    02  FILLER                         PIC X(50)
        VALUE SPACES.
    02  FILLER                         PIC X(6)
        VALUE 'PAGE  '.
    02  O-PAGE-NUMBER                  PIC 999
        VALUE 0.
    02  FILLER                         PIC XX
        VALUE SPACES.

01  REPORT-LINE-2.
    02  FILLER                         PIC X(63)
        VALUE SPACES.
    02  FILLER                         PIC X(6)
        VALUE 'REPORT'.
    02  FILLER                         PIC X(63)
        VALUE SPACES.

01  HEADING-LINE-1.
```

```
           02  FILLER                      PIC  X(4)
               VALUE SPACES.
           02  FILLER                      PIC  X(5)
               VALUE 'STORE'.
           02  FILLER                      PIC  X(4)
               VALUE SPACES.
           02  FILLER                      PIC  X(10)
               VALUE 'DEPARTMENT'.
           02  FILLER                      PIC  X(4)
               VALUE SPACES.
           02  FILLER                      PIC  X(8)
               VALUE 'SALESMAN'.
           02  FILLER                      PIC  X(9)
               VALUE SPACES.
           02  FILLER                      PIC  X(8)
               VALUE 'SALESMAN'.
           02  FILLER                      PIC  X(10)
               VALUE SPACES.
           02  FILLER                      PIC  X(12)
               VALUE 'YEAR TO DATE'.
           02  FILLER                      PIC  X(5)
               VALUE SPACES.
           02  FILLER                      PIC  X(7)
               VALUE 'CURRENT'.
           02  FILLER                      PIC  X(4)
               VALUE SPACES.
           02  FILLER                      PIC  X(10)
               VALUE 'COMMISSION'.
           02  FILLER                      PIC  X(5)
               VALUE SPACES.
           02  FILLER                      PIC  X(7)
               VALUE 'CURRENT'.
           02  FILLER                      PIC  X(6)
               VALUE SPACES.
           02  FILLER                      PIC  X(6)
               VALUE 'MONTHS'.
           02  FILLER                      PIC  X(8)
               VALUE SPACES.

       01  HEADING-LINE-2.
           02  FILLER                      PIC  X(4)
               VALUE SPACES.
           02  FILLER                      PIC  X(6)
               VALUE 'NUMBER'.
           02  FILLER                      PIC  X(18)
               VALUE SPACES.
           02  FILLER                      PIC  X(6)
               VALUE 'NUMBER'.
           02  FILLER                      PIC  X(12)
               VALUE SPACES.
           02  FILLER                      PIC  X(4)
               VALUE 'NAME'.
           02  FILLER                      PIC  X(16)
               VALUE SPACES.
           02  FILLER                      PIC  X(5)
               VALUE 'SALES'.
           02  FILLER                      PIC  X(9)
               VALUE SPACES.
           02  FILLER                      PIC  X(5)
               VALUE 'SALES'.
           02  FILLER                      PIC  X(8)
               VALUE SPACES.
           02  FILLER                      PIC  X(4)
               VALUE 'RATE'.
```

```
          02  FILLER                          PIC  X(7)
              VALUE SPACES.
          02  FILLER                          PIC  X(10)
              VALUE 'COMMISSION'.
          02  FILLER                          PIC  X(3)
              VALUE SPACES.
          02  FILLER                          PIC  X(8)
              VALUE 'EMPLOYED'.
          02  FILLER                          PIC  X(7)
              VALUE SPACES.

      01  VALID-DATA-LINE.
          02  FILLER                          PIC X(6)
              VALUE SPACES.
          02  O-STORE-NUMBER                  PIC Z9.
          02  FILLER                          PIC X(9)
              VALUE SPACES.
          02  O-DEPARTMENT                    PIC XX.
          02  FILLER                          PIC X(10)
              VALUE SPACES.
          02  O-SALESMAN-NUMBER               PIC ZZ9.
          02  FILLER                          PIC X(6)
              VALUE SPACES.
          02  O-SALESMAN-NAME                 PIC X(20).
          02  FILLER                          PIC X(6)
              VALUE SPACES.
          02  O-YEAR-TO-DATE-SALES            PIC Z(6).99.
          02  FILLER                          PIC X(5)
              VALUE SPACES.
          02  O-CURRENT-SALES                 PIC Z(6).99.
          02  FILLER                          PIC X(7)
              VALUE SPACES.
          02  O-COMMISSION-RATE               PIC .99.
          02  FILLER                          PIC X(7)
              VALUE SPACES.
          02  O-CURRENT-COMMISSION            PIC Z(5).99.
          02  FILLER                          PIC X(8)
              VALUE SPACES.
          02  O-MONTHS-EMPLOYED               PIC Z9.
          02  FILLER                          PIC X(10)
              VALUE SPACES.

      01  INVALID-DATA-LINE.
          02  O-BAD-DATA                      PIC  X(45).
          02  FILLER                          PIC  X(30)
              VALUE '     INVALID DATA ON THIS CARD'.
          02  FILLER                          PIC  X(57)
              VALUE SPACES.




      PROCEDURE DIVISION.


      PREPARE-PAYMENT-REPORT.
          OPEN INPUT CARD-FILE
              OUTPUT PRINT-FILE.
          READ CARD-FILE
              AT END MOVE 'NO' TO MORE-DATA-REMAINS-FLAG.

          IF MORE-DATA-REMAINS-FLAG = 'YES'
              PERFORM REPORT-HEADER-OUTPUT
              PERFORM HEADING-OUTPUT
```

```
            PERFORM COMMISSION-CALCULATION
                UNTIL MORE-DATA-REMAINS-FLAG = 'NO '.

        PERFORM CALCULATED-TOTALS-OUTPUT.
        CLOSE CARD-FILE
            PRINT-FILE.
        STOP RUN.


 *  CHECK VARIABLES TO SEE IF THEY CONTAIN VALID INFORMATION

 VALIDATION.
     IF I-STORE-NUMBER IS NUMERIC
       AND I-SALESMAN-NUMBER IS NUMERIC
       AND I-YEAR-TO-DATE-SALES IS NUMERIC
       AND I-CURRENT-SALES IS NUMERIC
       AND I-COMMISSION-RATE IS NUMERIC
       AND I-MONTHS-EMPLOYED IS NUMERIC
         MOVE 'YES' TO VALID-DATA-FLAG
     ELSE
         MOVE 'NO' TO VALID-DATA-FLAG.


 *  MOVE INPUT INFORMATION TO WORKING STORAGE
 *  VARIABLES

 DATA-MOVE.
     MOVE I-STORE-NUMBER TO W-STORE-NUMBER.
     MOVE I-DEPARTMENT TO W-DEPARTMENT.
     MOVE I-SALESMAN-NUMBER TO W-SALESMAN-NUMBER.
     MOVE I-YEAR-TO-DATE-SALES TO W-YEAR-TO-DATE-SALES.
     MOVE I-CURRENT-SALES TO W-CURRENT-SALES.
     MOVE I-COMMISSION-RATE TO W-COMMISSION-RATE.
     MOVE I-MONTHS-EMPLOYED TO W-MONTHS-EMPLOYED.

 CALCULATE-DEPARTMENT-BONUS.
     IF W-DEPARTMENT = '01' OR
        W-DEPARTMENT = '02'
         MOVE DEPT-1-OR-2 TO W-DEPARTMENT-BONUS
     ELSE IF W-DEPARTMENT = '06' OR
             W-DEPARTMENT = '09'
         MOVE DEPT-6-OR-9 TO W-DEPARTMENT-BONUS
     ELSE IF W-DEPARTMENT = '05' OR
             W-DEPARTMENT = '07'
         MOVE DEPT-5-OR-7 TO W-DEPARTMENT-BONUS
     ELSE IF W-DEPARTMENT = '04'
         MOVE DEPT-4 TO W-DEPARTMENT-BONUS
     ELSE
         MOVE DEPT-OTHER TO W-DEPARTMENT-BONUS.

 CALCULATE-VOLUME-BONUS.
     COMPUTE W-AVERAGE-MONTHLY-SALES ROUNDED =
       ( W-YEAR-TO-DATE-SALES + W-CURRENT-SALES )
       / W-MONTHS-EMPLOYED.
     IF W-AVERAGE-MONTHLY-SALES < 500
         MOVE LEVEL-1 TO W-VOLUME-BONUS
     ELSE IF W-AVERAGE-MONTHLY-SALES < 999.99
         MOVE LEVEL-2 TO W-VOLUME-BONUS
     ELSE IF W-AVERAGE-MONTHLY-SALES < 1999.99
         MOVE LEVEL-3 TO W-VOLUME-BONUS
     ELSE
         MOVE LEVEL-4 TO W-VOLUME-BONUS.

 COMMISSION-CALCULATION.
```

```
           PERFORM VALIDATION.

           IF VALID-DATA-FLAG = 'YES'
               PERFORM DATA-MOVE
               PERFORM CALCULATE-DEPARTMENT-BONUS
               PERFORM CALCULATE-VOLUME-BONUS
               COMPUTE W-CURRENT-COMMISSION ROUNDED = W-CURRENT-SALES *
                   ( W-COMMISSION-RATE + W-VOLUME-BONUS +
                   W-DEPARTMENT-BONUS )
               ADD W-YEAR-TO-DATE-SALES TO W-TOTAL-YEAR-TO-DATE-SALES
               ADD W-CURRENT-SALES TO W-TOTAL-CURRENT-SALES
               ADD W-CURRENT-COMMISSION TO W-TOTAL-CURRENT-COMMISSION
               PERFORM VALID-DATA-OUTPUT
           ELSE
               PERFORM INVALID-DATA-OUTPUT.

           READ CARD-FILE
               AT END MOVE 'NO' TO MORE-DATA-REMAINS-FLAG.

       VALID-DATA-OUTPUT.
           MOVE W-STORE-NUMBER TO O-STORE-NUMBER.
           MOVE W-DEPARTMENT TO O-DEPARTMENT.
           MOVE W-SALESMAN-NUMBER TO O-SALESMAN-NUMBER.
           MOVE I-SALESMAN-NAME TO O-SALESMAN-NAME.
           MOVE W-YEAR-TO-DATE-SALES TO O-YEAR-TO-DATE-SALES.
           MOVE W-CURRENT-SALES TO O-CURRENT-SALES.
           MOVE W-COMMISSION-RATE TO O-COMMISSION-RATE.
           MOVE W-CURRENT-COMMISSION TO O-CURRENT-COMMISSION.
           MOVE W-MONTHS-EMPLOYED TO O-MONTHS-EMPLOYED.
      *    MOVE I-SALESMAN-NAME TO O-SALESMAN-NAME.
           MOVE VALID-DATA-LINE TO LINE-RECORD.
           WRITE LINE-RECORD AFTER ADVANCING 1 LINES.
           ADD 1 TO LINE-COUNT.
           IF LINE-COUNT IS GREATER THAN 10
      *        MOVE 0 TO LINE-COUNT
               PERFORM REPORT-HEADER-OUTPUT
               PERFORM HEADING-OUTPUT.

       INVALID-DATA-OUTPUT.
           MOVE I-CARD-DATA TO O-BAD-DATA.
           MOVE INVALID-DATA-LINE TO LINE-RECORD.
           WRITE LINE-RECORD AFTER ADVANCING 1 LINES.
           ADD 1 TO LINE-COUNT.
           IF LINE-COUNT IS GREATER THAN 10
      *        MOVE 0 TO LINE-COUNT
               PERFORM REPORT-HEADER-OUTPUT
               PERFORM HEADING-OUTPUT.

       HEADING-OUTPUT.
           MOVE HEADING-LINE-1 TO LINE-RECORD.
           WRITE LINE-RECORD AFTER ADVANCING 1 LINES.
           MOVE HEADING-LINE-2 TO LINE-RECORD.
           WRITE LINE-RECORD AFTER ADVANCING 1 LINES.
           MOVE SPACES TO LINE-RECORD.
           WRITE LINE-RECORD AFTER ADVANCING 2 LINES.
           ADD 4 TO LINE-COUNT.

       CALCULATED-TOTALS-OUTPUT.
           MOVE W-TOTAL-YEAR-TO-DATE-SALES TO O-TOTAL-YEAR-TO-DATE-SALES
           MOVE W-TOTAL-CURRENT-SALES TO O-TOTAL-CURRENT-SALES.
           MOVE W-TOTAL-CURRENT-COMMISSION TO O-TOTAL-CURRENT-COMMISSION
           MOVE FINAL-TOTAL-LINE TO LINE-RECORD.
           WRITE LINE-RECORD AFTER ADVANCING 2 LINES.
```

```
REPORT-HEADER-OUTPUT.
    ADD 1 TO O-PAGE-NUMBER.
    MOVE REPORT-LINE-1 TO LINE-RECORD.
    WRITE LINE-RECORD AFTER ADVANCING TO-TOP-OF-PAGE.
    MOVE REPORT-LINE-2 TO LINE-RECORD.
    WRITE LINE-RECORD AFTER ADVANCING 1 LINES.
    MOVE SPACES TO LINE-RECORD.
    WRITE LINE-RECORD AFTER ADVANCING 3 LINES.
    MOVE 4 TO LINE-COUNT.
```

**Program A6**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINTMFS.
REMARKS.  THIS PROGRAM IS ADAPTED FROM YOURDAN'S "LEARNING
          TO PROGRAM IN STRUCTURED COBOL".
          (1)    THE PROGRAM AS PUBLISHED DID NOT WORK.  THE LAST
          PAIR OF APPLICATION CARDS WAS IGNORED.  IF THERE
          WAS NO LAST PAIR (EMPTY FILE) THE PROGRAM BOMBED.
          THIS ERROR WAS FIXED BY ADDING ANOTHER FILE-CONTROL
          FLAG AND ADDING LOGIC IN "B1-GET-A-PAIR..."
          (2)    THE NOTE ABOUT CHECKING PAIR VALIDITY
          IN PARAGRAPH "A2-UPDATE MASTER" SHOULD BE REPEATED
          IN THE ANALOGOUS PARAGRAPH "A4-ADD-REMAINING-CARDS".
          (3)    IF THE FIRST CARD IS INVALID, ITS LOG ENTRY
          WOULD HAVE BEEN WRITTEN BEFORE THE LOG FILE HEADER.
          (4)    THE PUBLISHED PROGRAM CONTAINED MUCH EXTRANEOUS
          CODE.  THE REASON FOR SOME OF THIS WAS THE FREE USE OF
          THE "COPY" VERB.  THESE PRODUCED MANY UNNECESSARY
          MUTANTS, AND HAVE BEEN COMMENTED OUT WITH "***".
          (5)    THE PROGRAM DID NOT DO ANYTHING SENSIBLE WHEN
          THE END-OF-FILE WAS ENCOUNTERED AFTER THE FIRST OF A
          PAIR OF CARDS.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  PRIME.
OBJECT-COMPUTER.  PRIME.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT APPLICATION-CARDS-FILE     ASSIGN TO INPUT1.
    SELECT UPDATE-LISTING             ASSIGN TO OUTPUT1.
    SELECT CREDIT-MASTER-OLD-FILE     ASSIGN TO INPUT2.
    SELECT CREDIT-MASTER-NEW-FILE     ASSIGN TO OUTPUT2.

DATA DIVISION.
FILE SECTION.

FD  APPLICATION-CARDS-FILE
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS NAME-ADDRESS-AND-PHONE-IN.
01  NAME-ADDRESS-AND-PHONE-IN.
    05  NAME-AND-ADDRESS-IN.
        10  NAME-IN                          PIC X(20).
***     10  ADDRESS-IN.
***         15  STREET-IN                    PIC X(20).
***         15  CITY-IN                      PIC X(13).
***         15  STATE-IN                     PIC XX.
***         15  ZIP-IN                       PIC X(5).
        10  ADDRESS-IN                       PIC X(40).
    05  PHONE-IN                         PIC X(11).
    05  FILLER                           PIC X.
    05  CHANGE-CODE-IN                   PIC XX.
    05  ACCT-NUM-IN1                     PIC 9(6).

FD  UPDATE-LISTING
    RECORD CONTAINS 132 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINT-LINE-OUT.
01  PRINT-LINE-OUT                       PIC X(132).

FD  CREDIT-MASTER-OLD-FILE
```

```
              RECORD CONTAINS 127 CHARACTERS
              LABEL RECORDS ARE STANDARD
              DATA RECORD IS CREDIT-MASTER-RECORD.
          01  CREDIT-MASTER-OLD-RECORD.
              05  ACCT-NUM-MAS-OLD                PIC 9(6).
      *** THE SUBFIELDS ARE NEVER REFERRRED TO IN THE PROGRAM
      ***     USE FILLER INSTEAD
      ***     05  NAME-AND-ADDRESS-MAS-OLD.
      ***         10  NAME-MAS-OLD                PIC X(20).
      ***         10  STREET-MAS-OLD              PIC X(20).
      ***         10  CITY-MAS-OLD                PIC X(13).
      ***         10  STATE-MAS-OLD               PIC XX.
      ***         10  ZIP-MAS-OLD                 PIC 9(5).
      ***     05  PHONE-MAS-OLD.
      ***         10  AREA-CODE-MAS-OLD           PIC 9(3).
      ***         10  NUMBER-MAS-OLD              PIC 9(7).
      *********************************************************
              05  FILLER                          PIC X(70).
      *** THE SUBFIELDS ARE NEVER REFERRED TO IN THE PROGRAM.
      ***     05  CREDIT-INFO-MAS-OLD.
      ***         10  SEX-MAS-OLD                 PIC X.
      ***         10  MARITAL-STATUS-MAS-OLD      PIC X.
      ***         10  NUMBER-DEPENS-MAS-OLD       PIC 99.
      ***         10  INCOME-HUNDREDS-MAS-OLD     PIC 9(3).
      ***         10  YEARS-EMPLOYED-MAS-OLD      PIC 99.
      ***         10  OWN-OR-RENT-MAS-OLD         PIC X.
      ***         10  MORGAGE-OR-RENTAL-MAS-OLD   PIC 9(3).
      ***         10  OTHER-PAYMENTS-MAS-OLD      PIC 9(3).
              05  CREDIT-INFO-MAS-OLD             PIC X(16).
              05  ACCOUNT-INFO-MAS-OLD.
      ***         10  DISCR-INCOME-MAS-OLD        PIC S9(3).
      ***         10  CREDIT-LIMIT-OLD            PIC 9(4).
                  10  FILLER                      PIC S9(3).
                  10  FILLER                      PIC 9(4).
                  10  CURRENT-BALANCE-OWING-OLD   PIC S9(6)V99.
              05  SPARE-CHARACTERS-OLD            PIC X(20).

          FD  CREDIT-MASTER-NEW-FILE
              RECORD CONTAINS 127 CHARACTERS
              LABEL RECORDS ARE STANDARD
              DATA RECORD IS CREDIT-MASTER-RECORD.
          01  CREDIT-MASTER-NEW-RECORD.
              05  ACCT-NUM-MAS-NEW                PIC 9(6).
      ***     05  NAME-AND-ADDRESS-MAS-NEW.
      ***         10  NAME-MAS-NEW                PIC X(20).
      ***         10  STREET-MAS-NEW              PIC X(20).
      ***         10  CITY-MAS-NEW                PIC X(13).
      ***         10  STATE-MAS-NEW               PIC XX.
      ***         10  ZIP-MAS-NEW                 PIC 9(5).
              05  NAME-AND-ADDRESS-MAS-NEW        PIC X(60).
              05  PHONE-MAS-NEW.
                  10  AREA-CODE-MAS-NEW           PIC 9(3).
                  10  NUMBR-MAS-NEW               PIC 9(7).
              05  CREDIT-INFO-MAS-NEW.
                  10  SEX-MAS-NEW                 PIC X.
                  10  MARITAL-STATUS-MAS-NEW      PIC X.
                  10  NUMBER-DEPENS-MAS-NEW       PIC 99.
                  10  INCOME-HUNDREDS-MAS-NEW     PIC 9(3).
                  10  YEARS-EMPLOYED-MAS-NEW      PIC 99.
                  10  OWN-OR-RENT-MAS-NEW         PIC X.
                  10  MORGAGE-OR-RENTAL-MAS-NEW   PIC 9(3).
                  10  OTHER-PAYMENTS-MAS-NEW      PIC 9(3).
              05  ACCOUNT-INFO-MAS-NEW.
                  10  DISCR-INCOME-MAS-NEW        PIC S9(3).
```

```
                10  CREDIT-LIMIT-MAS-NEW          PIC 9(4).
                10  CURRENT-BALANCE-OWING-NEW     PIC S9(6)V99.
            05  SPARE-CHARACTERS-NEW              PIC X(20).

       WORKING-STORAGE SECTION.

       01  CREDIT-INFORMATION-IN.
            05  CARD-TYPE-IN                      PIC X.
            05  ACCT-NUM-IN2                      PIC 9(6).
            05  FILLER                            PIC X.
            05  CREDIT-INFO-IN                    PIC X(22).
            05  FILLER                            PIC X(50).

       01  COMMON-WS.
            05  CARDS-LEFT                        PIC X(3).
            05  NEXT-CARD-THERE                   PIC X(3).
            05  OLD-MASTER-RECORDS-LEFT           PIC X(3).
            05  NEW-MASTER-RECORDS-LEFT           PIC X(3).
            05  FIRST-CARD                        PIC X(4).
            05  SECOND-CARD                       PIC X(4).
            05  ACCT-NUM-MATCH                    PIC X(4).
            05  PAIR-VALIDITY                     PIC X(4).

       01  LOG-HEADER-WSA1.
            05  FILLER                            PIC X(47) VALUE SPACES.
            05  FILLER                            PIC X(38)
                VALUE 'LOG OF ADDITIONS DELETIONS AND CHANGES'.
            05  FILLER                            PIC X(47) VALUE SPACES.

     ***01   HEADER-WSA5.
     ***      05  FILLER                            PIC X(51)   VALUE SPACES
     ***      05  TITLE                             PIC X(30)
     ***              VALUE 'CONTENTS OF CREDIT MASTER FILE'.
     ***      05  FILLER                            PIC X(51)   VALUE SPACES
       01  APPLICATION-DATA-WSB2.
            05  NAME-AND-ADDRESS-WS.
                10  NAME-WS                       PIC X(20).
     ***        10  ADDRESS-WS.
     ***            15  STREET-WS                 PIC X(20).
     ***            15  CITY-WS                   PIC X(13).
     ***            15  STATE-WS                  PIC XX.
     ***            15  ZIP-WS                    PIC X(5).
                10  ADDRESS-WS                    PIC X(40).
            05  PHONE-WS .
                10  AREA-CODE-WS                  PIC 9(3).
                10  NUMBR-WS                      PIC X(8).
            05  FILLER                            PIC X  VALUE SPACE.
            05  CHANGE-CODE-WS                    PIC XX.
            05  ACCT-NUM-WS                       PIC 9(6).
            05  CREDIT-INFO-WS.
                10  SEX-WS                        PIC X.
     **              88  MALE       VALUE 'M'.
     **              88  FEMALE     VALUE 'F'.
                10  FILLER                        PIC X.
                10  MARITAL-STATUS-WS             PIC X.
     **              88  SINGLE     VALUE 'S'.
     **              88  MARRIED    VALUE 'M'.
     **              88  DIVORCED   VALUE 'D'.
     **              88  WIDOWED    VALUE 'W'.
                10  FILLER                        PIC X.
                10  NUMBER-DEPENS-WS              PIC 9.
                10  FILLER                        PIC X.
                10  INCOME-HUNDREDS-WS            PIC 9(3).
                10  FILLER                        PIC X.
```

```
                10  YEARS-EMPLOYED-WS           PIC 99.
                10  FILLER                      PIC X.
                10  OWN-OR-RENT-WS              PIC X.
 **                 88  OWNED        VALUE 'O'.
 **                 88  RENTED       VALUE 'R'.
                10  FILLER                      PIC X.
                10  MORGAGE-OR-RENTAL-WS        PIC 9(3).
                10  FILLER                      PIC X.
                10  OTHER-PAYMENTS-WS           PIC 9(3).

     01  UPDATE-MESSAGE-AREA-WSB2.
         05  UPDATE-MESSAGE-AREA                PIC X(15).

     01  CREDIT-MASTER-PRINT-LINE.
         05  FILLER                             PIC X(4)   VALUE SPACES.
         05  CREDIT-MASTER-OUT                  PIC X(128).

     01  UPDATE-RECORD-PRINT-LINE.
         05  FILLER                             PIC X(4)  VALUE SPACES.
         05  APPLICATION-DATA-OUT               PIC X(102).
         05  FILLER                             PIC X(4)  VALUE SPACES.
         05  MESSAGE-AREA-OUT                   PIC X(15).

     01  DISCR-INCOME-CALC-FIELDS-WSC8.
         05  ANNUAL-INCOME-WS                   PIC 9(5).
         05  ANNUAL-TAX-WS                      PIC 9(5).
         05  TAX-RATE-WS                        PIC 9V99   VALUE 0.25.
         05  MONTHS-IN-YEAR                     PIC 99     VALUE 12.
         05  MONTHLY-NET-INCOME-WS              PIC 9(4).
         05  MONTHLY-PAYMENTS-WS                PIC 9(4).
         05  DISCR-INCOME-WS                    PIC S9(3).

     01  CREDIT-LIMIT-CALC-FIELDS-WSC9.
         05  CREDIT-FACTOR                      PIC 9.
         05  FACTOR1                            PIC 9     VALUE 1.
         05  FACTOR2                            PIC 9     VALUE 2.
         05  FACTOR3                            PIC 9     VALUE 3.
         05  FACTOR4                            PIC 9     VALUE 4.
         05  FACTOR5                            PIC 9     VALUE 5.
         05  CREDIT-LIMIT-WS                    PIC 9(4).
         05  UPPER-LIMIT-WS                     PIC 9(4)  VALUE 2500.
 *** NEVER USED
 *** 05  TOTAL-CREDIT-GIVEN-WS                  PIC 9(7).

     01  ASSEMBLE-TEL-NUM-WSD1.
         05  TEL-NUMBR-WITH-HYPHEN.
             10  EXCHANGE-IN                    PIC 9(3).
             10  FILLER                         PIC X.
             10  FOUR-DIGIT-NUMBR-IN            PIC 9(4).
         05  TEL-NUMBR-WITHOUT-HYPHEN.
             10  EXCHANGE                       PIC 9(3).
             10  FOUR-DIGIT-NUMBR               PIC 9(4).

     01  CARD-ERROR-LINE1-WS.
         05  FILLER                             PIC X(5)  VALUE SPACES.
         05  FILLER                             PIC X(12)
                 VALUE 'FIRST CARD  '.
         05  FIRST-CARD-ERR1                    PIC X(4).
         05  FILLER                             PIC XX  VALUE SPACES.
         05  NAME-ERR1                          PIC X(20).
         05  ADDRESS-ERR1                       PIC X(40).
         05  PHONE-ERR1                         PIC X(11).
         05  FILLER                             PIC X(3)    VALUE SPACES.
         05  ACCT-NUM-ERR1                      PIC 9(6).
```

```

 01  CARD-ERROR-LINE2-WS.
     05  FILLER                           PIC X(5)  VALUE SPACES.
     05  FILLER                           PIC X(12)
             VALUE  'SECOND CARD '.
     05  SECOND-CARD-ERR2                 PIC X(4).
     05  FILLER                           PIC X(2)    VALUE SPACES.
     05  CREDIT-INFO-ERR2                 PIC X(80).
     05  MESSAGE-ERR-LINE-2               PIC X(29)   VALUE SPACES.

 PROCEDURE DIVISION.

 A0-MAIN-BODY.
     PERFORM A1-INITIALIZE.
     PERFORM A2-UPDATE-MASTER
       UNTIL OLD-MASTER-RECORDS-LEFT = 'NO '
         OR CARDS-LEFT = 'NO '.
     IF CARDS-LEFT = 'NO '
*                                          THERE ARE MORE OLD MASTER REC
         PERFORM A3-COPY-REMAINING-OLD-MASTER
           UNTIL OLD-MASTER-RECORDS-LEFT = 'NO '
     ELSE
*                                          THERE ARE NO MORE CARDS, SO
         PERFORM A4-ADD-REMAINING-CARDS
           UNTIL CARDS-LEFT = 'NO '.
*******************************************************************
*    CODE TO LIST THE CONTENTS OF THE NEW MASTER HAS BEEN OMITTED.
*    IT WOULD HAVE REQUIRED CLOSING THE NEW MASTER AND REOPENING
*    IT FOR INPUT.  THIS IS BEYOND THE ABILITIES OF CMS.1
*    THE DELETION AMOUNTS TO ABOUT 20 LINES OF CODE.
*******************************************************************
     PERFORM A7-END-OF-JOB.
     STOP RUN.

 A1-INITIALIZE.
     OPEN INPUT      APPLICATION-CARDS-FILE
                     CREDIT-MASTER-OLD-FILE
          OUTPUT     CREDIT-MASTER-NEW-FILE
                     UPDATE-LISTING.
*** USELESS INITIALIZATIONS HAVE BEEN COMMENTED OUT
***  MOVE SPACES TO FIRST-CARD.
***  MOVE SPACES TO SECOND-CARD.
***  MOVE SPACES TO ACCT-NUM-MATCH.
***  MOVE SPACES TO PAIR-VALIDITY.
***  MOVE ZEROES TO ANNUAL-INCOME-WS.
***  MOVE ZEROES TO ANNUAL-TAX-WS.
***  MOVE ZEROES TO MONTHLY-NET-INCOME-WS.
***  MOVE ZEROES TO MONTHLY-PAYMENTS-WS.
***  MOVE ZEROES TO DISCR-INCOME-WS.
***  MOVE ZEROES TO CREDIT-FACTOR.
***  MOVE ZEROES TO CREDIT-LIMIT-WS.
***  MOVE ZEROES TO TOTAL-CREDIT-GIVEN-WS.
     MOVE 'YES' TO CARDS-LEFT.
     MOVE 'YES' TO NEXT-CARD-THERE.
     MOVE 'YES' TO OLD-MASTER-RECORDS-LEFT.
**  THE FOLLOWING STATEMENT WAS MOVED HERE FROM THE END OF THE
**  PARAGRAPH, SO THAT THE HEADER WOULD BE WRITTEN BEFORE THE
**  FIRST LOG RECORD, IF THE FIRST CARD PAIR IS INVALID.
     WRITE PRINT-LINE-OUT FROM LOG-HEADER-WSA1
                              AFTER ADVANCING 3 LINES.
     READ APPLICATION-CARDS-FILE
         AT END MOVE 'NO ' TO NEXT-CARD-THERE.
     PERFORM B1-GET-A-PAIR-OF-CARDS-INTO-WS THRU B1-EXIT.
* FIRST PAIR OF CARDS IN WS: FIRST CARD OF SECOND PAIR IN BUFFER
```

```
       READ CREDIT-MASTER-OLD-FILE
           AT END MOVE 'NO ' TO OLD-MASTER-RECORDS-LEFT.
 * FIRST OLD MASTER RECORD IS IN BUFFER

  A2-UPDATE-MASTER.
 * BEFORE COMPARING THE UPDATE WITH THE MASTER, WE MUST CHECK
 * THAT WE HAVE A VALID PAIR OF CARDS - IF YOUR PROGRAM DOES
 * NOT MAKE THIS TEST, IT WILL ONLY WORK WITH VALID PAIRS OF
 * CARDS.
       IF  PAIR-VALIDITY = 'BAD '
           PERFORM B1-GET-A-PAIR-OF-CARDS-INTO-WS THRU B1-EXIT
       ELSE IF ACCT-NUM-WS IS GREATER THAN ACCT-NUM-MAS-OLD
 *                  ACCT-NUM-WS IS CARD ACCOUNT NUMBER
           MOVE CREDIT-MASTER-OLD-RECORD TO
               CREDIT-MASTER-NEW-RECORD
           WRITE CREDIT-MASTER-NEW-RECORD
           READ CREDIT-MASTER-OLD-FILE
               AT END MOVE 'NO ' TO OLD-MASTER-RECORDS-LEFT
       ELSE IF ACCT-NUM-WS = ACCT-NUM-MAS-OLD
           PERFORM B2-CHANGE-OR-DELETE-MASTER
           PERFORM B1-GET-A-PAIR-OF-CARDS-INTO-WS THRU B1-EXIT
           READ CREDIT-MASTER-OLD-FILE
               AT END MOVE 'NO ' TO OLD-MASTER-RECORDS-LEFT
       ELSE
 *                                 ACCT-NUM-WS IS LESS THAN
 *                                 ACCT-NUM-MAS-OLD
           PERFORM B3-ADD-NEW-MASTER
           PERFORM B1-GET-A-PAIR-OF-CARDS-INTO-WS THRU B1-EXIT.

  A3-COPY-REMAINING-OLD-MASTER.
      MOVE CREDIT-MASTER-OLD-RECORD TO
          CREDIT-MASTER-NEW-RECORD
      WRITE CREDIT-MASTER-NEW-RECORD.
      READ CREDIT-MASTER-OLD-FILE
          AT END MOVE 'NO ' TO OLD-MASTER-RECORDS-LEFT.

  A4-ADD-REMAINING-CARDS.
      IF PAIR-VALIDITY = 'BAD ' NEXT SENTENCE
      ELSE PERFORM B3-ADD-NEW-MASTER.
      PERFORM B1-GET-A-PAIR-OF-CARDS-INTO-WS THRU B1-EXIT.

  A7-END-OF-JOB.
      CLOSE APPLICATION-CARDS-FILE
            CREDIT-MASTER-OLD-FILE
            CREDIT-MASTER-NEW-FILE
            UPDATE-LISTING.

  B1-GET-A-PAIR-OF-CARDS-INTO-WS.
      IF NEXT-CARD-THERE = 'NO '
         MOVE 'NO ' TO CARDS-LEFT
         GO TO B1-EXIT.
      PERFORM C1-EDIT-FIRST-CARD.
      PERFORM C2-MOVE-FIRST-CARD-TO-WS.
      READ APPLICATION-CARDS-FILE INTO CREDIT-INFORMATION-IN
          AT END MOVE 'NO ' TO CARDS-LEFT,
                 MOVE SPACES TO  CREDIT-INFORMATION-IN
                                 ACCT-NUM-MATCH
                 MOVE 'NONE' TO SECOND-CARD
                 PERFORM C4-FLUSH-CARDS-TO-ERROR-LINES
                 GO TO B1-EXIT.
      PERFORM C3-EDIT-SECOND-CARD.
      IF (FIRST-CARD = 'GOOD')
          AND (SECOND-CARD = 'GOOD')
          AND (ACCT-NUM-MATCH = 'GOOD')
```

```
                 MOVE 'GOOD' TO PAIR-VALIDITY
                 MOVE CREDIT-INFO-IN TO CREDIT-INFO-WS
         ELSE
             MOVE 'BAD ' TO PAIR-VALIDITY
             PERFORM C4-FLUSH-CARDS-TO-ERROR-LINES.
         READ APPLICATION-CARDS-FILE
             AT END MOVE 'NO ' TO NEXT-CARD-THERE.

     B1-EXIT.  EXIT.

     B2-CHANGE-OR-DELETE-MASTER.
         IF  CHANGE-CODE-WS = 'CH'
             PERFORM C5-MERGE-UPDATE-WITH-OLD-MAST
             MOVE 'RECORD CHANGED' TO UPDATE-MESSAGE-AREA
             PERFORM C6-LOG-ACTION
             WRITE CREDIT-MASTER-NEW-RECORD
         ELSE IF CHANGE-CODE-WS = 'DE'
*                                    CHECK IF DELETE IS VALID
                 IF  CREDIT-INFO-WS IS EQUAL TO SPACES
                     MOVE 'RECORD DELETED' TO UPDATE-MESSAGE-AREA
                     PERFORM C6-LOG-ACTION
                 ELSE
                     MOVE 'REC NOT DELETED' TO UPDATE-MESSAGE-AREA
                     MOVE CREDIT-MASTER-OLD-RECORD TO
                     CREDIT-MASTER-NEW-RECORD
                     PERFORM C6-LOG-ACTION
                     WRITE CREDIT-MASTER-NEW-RECORD
         ELSE
             MOVE 'BAD CHANGE CODE' TO UPDATE-MESSAGE-AREA
             MOVE CREDIT-MASTER-OLD-RECORD TO CREDIT-MASTER-NEW-RECORD
             PERFORM C6-LOG-ACTION
             WRITE CREDIT-MASTER-NEW-RECORD.

     B3-ADD-NEW-MASTER.
         PERFORM C8-CALC-DISCRETNRY-INCOME.
         PERFORM C9-CALC-CREDIT-LIMIT.
         PERFORM C10-ASSEMBLE-NEW-MASTER-RECORD.
         MOVE 'RECORD ADDED ' TO UPDATE-MESSAGE-AREA.
         PERFORM C6-LOG-ACTION.
         WRITE CREDIT-MASTER-NEW-RECORD.

     C1-EDIT-FIRST-CARD.
         MOVE 'GOOD' TO FIRST-CARD.
         IF NAME-IN IS EQUAL TO SPACES
             MOVE '*** NAME MISSING ***' TO NAME-IN
             MOVE 'BAD ' TO FIRST-CARD.
         IF ADDRESS-IN IS EQUAL TO SPACES
             MOVE '*** ADDRESS MISSING ***' TO ADDRESS-IN
             MOVE 'BAD ' TO FIRST-CARD.
         IF PHONE-IN IS EQUAL TO SPACES
             MOVE 'NO PHONE **' TO PHONE-IN
             MOVE 'BAD ' TO FIRST-CARD.

     C2-MOVE-FIRST-CARD-TO-WS.
         MOVE NAME-IN TO NAME-WS.
         MOVE ADDRESS-IN TO ADDRESS-WS.
         MOVE PHONE-IN TO PHONE-WS.
         MOVE CHANGE-CODE-IN TO CHANGE-CODE-WS.
         MOVE ACCT-NUM-IN1 TO ACCT-NUM-WS.

     C3-EDIT-SECOND-CARD.
         MOVE 'GOOD' TO SECOND-CARD.
         MOVE 'GOOD' TO ACCT-NUM-MATCH.
         IF  CARD-TYPE-IN IS NOT EQUAL TO 'C'
```

```
                 MOVE 'BAD ' TO SECOND-CARD.
             IF  ACCT-NUM-IN2 IS NOT EQUAL TO ACCT-NUM-WS
                 MOVE 'BAD ' TO ACCT-NUM-MATCH.

       C4-FLUSH-CARDS-TO-ERROR-LINES.
             MOVE FIRST-CARD TO FIRST-CARD-ERR1.
             MOVE NAME-WS TO NAME-ERR1.
             MOVE ADDRESS-WS TO ADDRESS-ERR1.
             MOVE PHONE-WS TO PHONE-ERR1.
             MOVE ACCT-NUM-WS TO ACCT-NUM-ERR1.
             MOVE SECOND-CARD TO SECOND-CARD-ERR2.
      **      MOVE CREDIT-INFO-WS TO CREDIT-INFO-ERR2.
      **    THE PREVIOUS LINE WAS IN ERROR (BY A SINGLE MUTATION) IN THE
      **    PUBLISHED PROGRAM.  THE CORRECT STATEMENT IS:
             MOVE CREDIT-INFO-IN TO CREDIT-INFO-ERR2.
             IF  ACCT-NUM-MATCH = 'BAD '
                 MOVE 'ACCOUNT NUMBERS DO NOT MATCH'
                                 TO MESSAGE-ERR-LINE-2
             ELSE
                 MOVE SPACES TO MESSAGE-ERR-LINE-2.
      ***    MOVE SPACES TO PRINT-LINE-OUT.
             WRITE PRINT-LINE-OUT FROM CARD-ERROR-LINE1-WS
                     AFTER ADVANCING 3 LINES.
      ***    MOVE SPACES TO PRINT-LINE-OUT.
             WRITE PRINT-LINE-OUT FROM CARD-ERROR-LINE2-WS
                     AFTER ADVANCING 1 LINES.


       C5-MERGE-UPDATE-WITH-OLD-MAST.
             MOVE ACCT-NUM-MAS-OLD TO ACCT-NUM-MAS-NEW.
             MOVE NAME-AND-ADDRESS-WS TO NAME-AND-ADDRESS-MAS-NEW.
             MOVE AREA-CODE-WS TO AREA-CODE-MAS-NEW.
             PERFORM D1-REMOVE-HYPHEN-FROM-TEL-NUM.
      * THE SECOND INPUT CARD HAS CREDIT DATA, IF THIS HAS TO BE
      * UPDATED THEN THE DISCRETIONARY INCOME CALC HAS TO BE RUN
             IF CREDIT-INFO-WS IS EQUAL TO SPACES
                 MOVE CREDIT-INFO-MAS-OLD TO CREDIT-INFO-MAS-NEW
                 MOVE ACCOUNT-INFO-MAS-OLD TO ACCOUNT-INFO-MAS-NEW
             ELSE
                 PERFORM C8-CALC-DISCRETNRY-INCOME
                 PERFORM C9-CALC-CREDIT-LIMIT
                 MOVE SEX-WS                   TO SEX-MAS-NEW
                 MOVE MARITAL-STATUS-WS        TO MARITAL-STATUS-MAS-NEW
                 MOVE NUMBER-DEPENS-WS         TO NUMBER-DEPENS-MAS-NEW
                 MOVE INCOME-HUNDREDS-WS       TO INCOME-HUNDREDS-MAS-NEW
                 MOVE YEARS-EMPLOYED-WS        TO YEARS-EMPLOYED-MAS-NEW
                 MOVE OWN-OR-RENT-WS           TO OWN-OR-RENT-MAS-NEW
                 MOVE MORGAGE-OR-RENTAL-WS     TO MORGAGE-OR-RENTAL-MAS-NEW
                 MOVE OTHER-PAYMENTS-WS        TO OTHER-PAYMENTS-MAS-NEW
                 MOVE DISCR-INCOME-WS          TO DISCR-INCOME-MAS-NEW
                 MOVE CREDIT-LIMIT-WS          TO CREDIT-LIMIT-MAS-NEW.
             MOVE CURRENT-BALANCE-OWING-OLD TO CURRENT-BALANCE-OWING-NEW.
             MOVE SPARE-CHARACTERS-OLD TO SPARE-CHARACTERS-NEW.

       C6-LOG-ACTION.
             IF CHANGE-CODE-WS = 'CH'
      *                               WRITE OLD TAPE RECORD
      *                               WRITE CARD CONTENTS & MESSAGE
      *                               WRITE NEW TAPE RECORD
      ***        MOVE SPACES TO CREDIT-MASTER-PRINT-LINE
                 MOVE CREDIT-MASTER-OLD-RECORD TO CREDIT-MASTER-OUT
                 WRITE PRINT-LINE-OUT FROM CREDIT-MASTER-PRINT-LINE
                             AFTER ADVANCING 3 LINES
      ***        MOVE SPACES TO UPDATE-RECORD-PRINT-LINE
```

```
               MOVE APPLICATION-DATA-WSB2 TO APPLICATION-DATA-OUT
               MOVE UPDATE-MESSAGE-AREA TO MESSAGE-AREA-OUT
               WRITE PRINT-LINE-OUT FROM UPDATE-RECORD-PRINT-LINE
                         AFTER ADVANCING 1 LINES
     ***       MOVE SPACES TO CREDIT-MASTER-PRINT-LINE
               MOVE CREDIT-MASTER-NEW-RECORD TO CREDIT-MASTER-OUT
               WRITE PRINT-LINE-OUT FROM CREDIT-MASTER-PRINT-LINE
                         AFTER ADVANCING 1 LINES
           ELSE  IF CHANGE-CODE-WS = 'DE'
     *                                  WRITE OLD TAPE RECORD
     *                                  WRITE CARD CONTENTS & MESSAGE
     ***       MOVE SPACES TO CREDIT-MASTER-PRINT-LINE
               MOVE CREDIT-MASTER-OLD-RECORD TO CREDIT-MASTER-OUT
               WRITE PRINT-LINE-OUT FROM CREDIT-MASTER-PRINT-LINE
                         AFTER ADVANCING 3 LINES
     ***       MOVE SPACES TO UPDATE-RECORD-PRINT-LINE
               MOVE APPLICATION-DATA-WSB2 TO APPLICATION-DATA-OUT
               MOVE UPDATE-MESSAGE-AREA TO MESSAGE-AREA-OUT
               WRITE PRINT-LINE-OUT FROM UPDATE-RECORD-PRINT-LINE
                         AFTER ADVANCING 1 LINES
           ELSE IF CHANGE-CODE-WS = '  '
     *                                  WRITE CARDS FOR ADDITION
     *                                  WRITE NEW TAPE RECORD
     ***       MOVE SPACES TO UPDATE-RECORD-PRINT-LINE
               MOVE APPLICATION-DATA-WSB2 TO APPLICATION-DATA-OUT
               MOVE UPDATE-MESSAGE-AREA TO MESSAGE-AREA-OUT
               WRITE PRINT-LINE-OUT FROM UPDATE-RECORD-PRINT-LINE
                         AFTER ADVANCING 3 LINES
     ***       MOVE SPACES TO CREDIT-MASTER-PRINT-LINE
               MOVE CREDIT-MASTER-NEW-RECORD TO CREDIT-MASTER-OUT
               WRITE PRINT-LINE-OUT FROM CREDIT-MASTER-PRINT-LINE
                         AFTER ADVANCING 1 LINES

           ELSE
     *                                  WRITE CARD CONTENTS & MESSAGE
               MOVE APPLICATION-DATA-WSB2 TO APPLICATION-DATA-OUT
               MOVE UPDATE-MESSAGE-AREA TO MESSAGE-AREA-OUT
               WRITE PRINT-LINE-OUT FROM UPDATE-RECORD-PRINT-LINE
                         AFTER ADVANCING 3 LINES.

       C8-CALC-DISCRETNRY-INCOME.
           COMPUTE ANNUAL-INCOME-WS = INCOME-HUNDREDS-WS * 100.
           COMPUTE ANNUAL-TAX-WS    = ANNUAL-INCOME-WS * TAX-RATE-WS.
           COMPUTE MONTHLY-NET-INCOME-WS ROUNDED
               = (ANNUAL-INCOME-WS - ANNUAL-TAX-WS) / MONTHS-IN-YEAR.
           COMPUTE MONTHLY-PAYMENTS-WS = MORGAGE-OR-RENTAL-WS
                                + OTHER-PAYMENTS-WS.
           COMPUTE DISCR-INCOME-WS = MONTHLY-NET-INCOME-WS
                                   - MONTHLY-PAYMENTS-WS
               ON SIZE ERROR MOVE 999 TO DISCR-INCOME-WS.
     *   DISCRETIONARY INCOMES OVER $999 PER MONTH ARE SET AT $999.

       C9-CALC-CREDIT-LIMIT.
     *     MARRIED?           Y Y Y Y N N N N   THIS DECISION TABLE     *
     *     OWNED?             Y Y N N Y Y N N   SETS OUT COMPANY POLICY *
     *     2 OR MORE YEARS?   Y N Y N Y N Y N   FOR DETERMINING CREDIT  *
     *     --------------------------------   LIMIT FROM DISCRETIONARY*
     *     CREDIT   FACTOR1                 X X   INCOME. FACTOR1 ETC ARE *
     *     LIMIT          2         X   X         SET UP IN WSC9.         *
     *     MULTIPLE       3           X                                   *
     *     OF DISCR.      4     X X                                       *
     *     INCOME         5   X                                           *
           IF MARITAL-STATUS-WS = 'M'
               IF OWN-OR-RENT-WS = 'O'
```

```
                IF  YEARS-EMPLOYED-WS IS NOT LESS THAN 02
                    MOVE FACTOR5 TO CREDIT-FACTOR
                ELSE
                    MOVE FACTOR4 TO CREDIT-FACTOR
            ELSE
                IF  YEARS-EMPLOYED-WS IS NOT LESS THAN 02
                    MOVE FACTOR4 TO CREDIT-FACTOR
                ELSE
                    MOVE FACTOR2 TO CREDIT-FACTOR
        ELSE
            IF OWN-OR-RENT-WS = 'O'
                IF  YEARS-EMPLOYED-WS IS NOT LESS THAN 02
                    MOVE FACTOR3 TO CREDIT-FACTOR
                ELSE
                    MOVE FACTOR2 TO CREDIT-FACTOR
            ELSE
                MOVE FACTOR1 TO CREDIT-FACTOR.
        COMPUTE CREDIT-LIMIT-WS = DISCR-INCOME-WS * CREDIT-FACTOR.
        IF CREDIT-LIMIT-WS IS GREATER THAN UPPER-LIMIT-WS
            MOVE UPPER-LIMIT-WS TO CREDIT-LIMIT-WS.
***     ADD CREDIT-LIMIT-WS TO TOTAL-CREDIT-GIVEN-WS.

    C10-ASSEMBLE-NEW-MASTER-RECORD.
        MOVE ACCT-NUM-WS TO ACCT-NUM-MAS-NEW.
        MOVE NAME-AND-ADDRESS-WS TO NAME-AND-ADDRESS-MAS-NEW.
        MOVE AREA-CODE-WS TO AREA-CODE-MAS-NEW.
        PERFORM D1-REMOVE-HYPHEN-FROM-TEL-NUM.
        MOVE SEX-WS                TO SEX-MAS-NEW
        MOVE MARITAL-STATUS-WS     TO MARITAL-STATUS-MAS-NEW
        MOVE NUMBER-DEPENS-WS      TO NUMBER-DEPENS-MAS-NEW
        MOVE INCOME-HUNDREDS-WS    TO INCOME-HUNDREDS-MAS-NEW
        MOVE YEARS-EMPLOYED-WS     TO YEARS-EMPLOYED-MAS-NEW
        MOVE OWN-OR-RENT-WS        TO OWN-OR-RENT-MAS-NEW
        MOVE MORGAGE-OR-RENTAL-WS  TO MORGAGE-OR-RENTAL-MAS-NEW
        MOVE OTHER-PAYMENTS-WS     TO OTHER-PAYMENTS-MAS-NEW.
        MOVE DISCR-INCOME-WS TO DISCR-INCOME-MAS-NEW.
        MOVE CREDIT-LIMIT-WS TO CREDIT-LIMIT-MAS-NEW.
        MOVE ZEROES TO CURRENT-BALANCE-OWING-NEW.
        MOVE SPACES TO SPARE-CHARACTERS-NEW.

    D1-REMOVE-HYPHEN-FROM-TEL-NUM.
        MOVE NUMBR-WS TO TEL-NUMBR-WITH-HYPHEN
        MOVE EXCHANGE-IN TO EXCHANGE
        MOVE FOUR-DIGIT-NUMBR-IN TO FOUR-DIGIT-NUMBR
        MOVE TEL-NUMBR-WITHOUT-HYPHEN TO NUMBR-MAS-NEW.
```

# Appendix B

**Program B1:**

The first program is written in an Algol dialect and initially appeared in a paper by Henderson and Snowden [Henderson, 1972]. Its intent is to read and process a string of characters that represent a sequence of telegrams, where a telegram is any string terminated by the keywords "ZZZZ ZZZZ." The program scans for words longer than a fixed limit and isolates and prints each telegram along with a count of the number of words it contains, plus an indication of the presence or absence of over-length words. The program has also been studied in Ledgard [Ledgard, 1973] and Gerhart and Yelowitz [Gerhart, 1976]. The program contains the following loop, which is intended to insure that blank characters are skipped and that following the loop the variable LETTER contains a non-blank character.

```
WHILE input ≠ emptystring AND FIRST(input) = ''
  DO input := REST(input);
IF input = emptystring THEN input = READ + '';
LETTER = FIRST(input);
```

The WHILE loop terminates either on an empty string or on a non-blank character. If it terminates on an empty string and the first character in the buffer loaded by the READ instruction is blank, LETTER can contain a blank character.

When this program is translated into Fortran and executed, the error is not necessarily caught. The reason for this failure is not so much a failure of mutation testing as it is of Fortran. Algol treats strings as a basic type, whereas in Fortran they are simulated by arrays of integers. The fact that strings are basic to

Algol means that if we were constructing a mutation system for Algol instead of Fortran we would have to consider a different set of mutant operators. A natural operator one would consider can be explained by noting that blanks play a role in string processing programs analogous to that played by zero in numbers. Hence we might hypothesize a "blank push" operator similar to ZPUSH. If we had such an operator, an attempt to force the expression FIRST(input) to blank would certainly reveal the error.

**Program B2:**

The second program appears in a paper by Wirth describing the language PL-360 [Wirth, 1968]. It is intended to take a vector of N numbers and sort them into decreasing order. It was also studied by Gerhart and Yelowitz [Gerhart, 1976]. As the outer loop is incremented over the list of elements, the inner loop is designed to find the maximum of the remaining elements and set register R3 to the index of this maximum. If the position set in the outer loop is indeed the maximum, then R3 will have an incorrect value and the three assignment statements ending the loop will give erroneous results.

```
Sort(R4)
For R1 = 0 by 4 to N begin
  R0 := a(R1)
  for R2 = R1 + 4 by 4 to N begin
    if a(R2) > R0 then begin
      R0 := a(R2)
      R3 := R2
    end
  end
  R2 := a(R1)
  a(R1) := R0
  a(R3) := R2
```

There are three mutants that cannot be eliminated without discovering this error. The first two change the statement R0 := A(R1) into R0 := A(R1)-1 and R0 := -ABS(A(R1)). The third mutant changes the statement into A(R1) := A(R3). We leave it as an exercise to verify that none of these mutants can be eliminated without discovering the error.

**Program B3:**

The third program is written in Fortran and computes the total, average, minimum, maximum, and standard deviation for each variable in an observation matrix. The program is adapted from the IBM scientific subroutines package [IBM, 1966]. It was analyzed and three artificial errors were inserted in a study by Gould and Drongowski [Gould, 1974]]. As in the study by Howden [Howden, 1978] we considered only one of these errors. It occurs in a loop that computes standard deviations. The program has the statement

```
SD(I) = SQRT(ABS((SD(I) - (TOTAL(I)*TOTAL(I))/SCNT)/SCNT - 1
```

A pair of parentheses has been left off the final SCNT - 1 expression. Let x stand for the quantity

```
ABS(SD(I) - (TOTAL(I)*TOTAL(I))/SCNT)
```

The correct standard deviation is SQRT(X/(SCNT-1)). The only way this can be made zero is for X to be zero. But the program containing the error computes the standard deviation as SQRT(1-X/SCNT). If X is zero this quantity is 1; hence the standard deviation is wrong.

Or if the incorrect expression is forced to be zero, then the correct standard deviation should be greater than one. Hence by forcing the standard deviation in this line to be zero the error is easily revealed.

**Program B4:**

The fourth program appeared in an article by Geller in the <u>Communications of the ACM</u> [Geller, 1978]. The program contains a predicate that decides whether a year is a leap year. In the paper this predicate is given as

```
          ((YEAR REM 4 = 0) OR
(YEAR REM 100 = 0 AND YEAR REM 400 = 0))
```

when the correct predicate is

```
((YEAR REM 4 = 0 AND YEAR REM 100 ≠ 0) OR
           (YEAR REM 400 = 0))
```

If YEAR is divisible by 400 then it must also be divisible by 100. In the incorrect predicate, therefore, the second part of the OR clause is true if and only if YEAR REM 400 is true. If a branch analysis method attempts to follow all the "hidden paths" [DeMillo, 1978a], the error will be discovered when an attempt is made to make YEAR REM 400 true and YEAR REM 100 false. With mutation analysis the error is discovered when we replace YEAR REM 100 with TRUE.

**Program B5:**

The fifth program computes the Euclidean greatest common divisor of a vector of integers. It appeared in an article by Bradley in the Communications of the ACM [Bradley, 1970]. The program contains the following four errors: (1) If the last input number is the only non-zero number and it is negative, then the greatest common divisor returned is negative. (2) If the greatest common divisor is not 1, then a loop index is used after the loop has completed normally, which is an error according to the Fortran standard. (3,4) There are two DO loops for which it is possible to construct data so that the upper limit is less than the lower limit, which causes the program to produce incorrect results since Fortran DO loops always execute at least once. None of the errors is caught using branch analysis. All are caught with mutation analysis.

The next three programs are adapted from the IBM Scientific Subroutines Package [IBM, 1966]. In each program three errors were artificially inserted in a study conducted by Gould and Drongowski [Gould, 1974].

**Program B6:**

The first program computes the first four moments of a vector of observations. One of the errors would be detected using branch analysis, the other two can be overlooked. All three errors would be discovered using mutation analysis.

**Program B7:**

The second program computes statistics from an observation table. Again, one error would be discovered using branch analysis but all three errors are discovered with mutation analysis.

**Program B8:**

The third program computes correlation coefficients. Two of the errors are detected with branch analysis; all three are detected with mutation analysis.

**Program B9:**

The next program takes three sides of a triangle and decides whether it is isosceles, scalene, or equilateral. It first appeared in a paper by Brown and Lipow [Brown, 1975]. Lipton and Sayward [Lipton, 1978] describe a bug where two occurrences of the constant 2 are replaced with the variable k. This bug is very subtle, but it can be detected with the test case 6,3,3. Neither branch analysis nor mutation analysis would force the discovery of this error.

**Program B10:**

The tenth program is the FIND program from an article by C.A.R. Hoare [Hoare, 1961]. The bug has been studied by the group developing the SELECT symbolic execution system [Boyer, 1975]. The bug is very subtle and neither branch analysis nor mutation analysis would guarantee its discovery. This bug was, however, easily discovered by mutation analysis (in the normal debugging situation) during some early experiments on the coupling effect [DeMillo, 1978a].

**Program B11:**

This program, also written in Algol, appeared in a paper by Naur [Nauer, 1969] and has also been studied widely [Foster, 1978], [Gerhart, 1976], [Goodenough, 1975]. The program is intended to read a string of characters consisting of words separated by blanks or newline characters or both, and to output as many words as possible with a blank between every pair of words. There is a fixed limit on the size of each output line, and no word can be broken between two lines. The version studied here is that of Gerhart and Yelowitz [Gerhart, 1976], containing five errors. Three of these (1, 3, and 4 in the numbering of [Gerhart, 1976]) are caught by mutation analysis.

**Program B12:**

This program maintains a stack. The user can select to enter data on the stack (PUSH), remove information from the stack (POP), examine the topmost stack element (TOP), or initialize the stack (CLEAR).

## Appendix C

LISTING THE PROGRAM UNIT "MOVENW " WITH SPECIFIED EQUIV MUTANTS

```
        SUBROUTINE MOVENW(SOURCE,SLEN,DEST,DLEN)
        INTEGER MLEN, K, SUB2, SUB1, LOOPHI, I, IHI, IER
        INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
        CHAR MEMORY(425)
        INTEGER DLEN, DEST, SLEN, SOURCE
        INPUT OUTPUT IER, MEMORY
        INPUT DLEN, DEST, SLEN, SOURCE
        MLEN = DLEN                                               1

$755$   MLEN = ABS DLEN
$757$   MLEN = ZPUSH DLEN

        IF(SLEN .LT. MLEN) MLEN = SLEN                            2  3

$43$    IF(SLEN .LT. DLEN) MLEN = SLEN
$630$   IF(-- SLEN .LT. MLEN) MLEN = SLEN
$632$   IF(SLEN .LT. ++ MLEN) MLEN = SLEN
$727$   IF(SLEN .LE. MLEN) MLEN = SLEN
$758$   IF(ABS SLEN .LT. MLEN) MLEN = SLEN
$760$   IF(ZPUSH SLEN .LT. MLEN) MLEN = SLEN
$761$   IF(SLEN .LT. ABS MLEN) MLEN = SLEN
$763$   IF(SLEN .LT. ZPUSH MLEN) MLEN = SLEN
$764$   IF(SLEN .LT. MLEN) MLEN = ABS SLEN
$766$   IF(SLEN .LT. MLEN) MLEN = ZPUSH SLEN

        LOOPHI = (DEST + MLEN) - 1                                4

$767$   LOOPHI = (ABS DEST + MLEN) - 1
$769$   LOOPHI = (ZPUSH DEST + MLEN) - 1
$770$   LOOPHI = (DEST + ABS MLEN) - 1
$772$   LOOPHI = (DEST + ZPUSH MLEN) - 1
$773$   LOOPHI = ABS (DEST + MLEN) - 1
$775$   LOOPHI = ZPUSH (DEST + MLEN) - 1
$776$   LOOPHI = ABS ((DEST + MLEN) - 1)
$778$   LOOPHI = ZPUSH ((DEST + MLEN) - 1)

        SUB2 = SOURCE - 1                                         5

$779$   SUB2 = ABS SOURCE - 1
$781$   SUB2 = ZPUSH SOURCE - 1
$782$   SUB2 = ABS (SOURCE - 1)
$784$   SUB2 = ZPUSH (SOURCE - 1)

        DO 20 SUB1=DEST, LOOPHI                                   6

$785$   DO 20 SUB1=ABS DEST, LOOPHI
$787$   DO 20 SUB1=ZPUSH DEST, LOOPHI
```

```
$788$  DO 20 SUB1=DEST, ABS LOOPHI
$790$  DO 20 SUB1=DEST, ZPUSH LOOPHI
$892$  FOR 20 SUB1=DEST, LOOPHI

       SUB2 = SUB2 + 1                                              7

$791$  SUB2 = ABS SUB2 + 1
$793$  SUB2 = ZPUSH SUB2 + 1
$794$  SUB2 = ABS (SUB2 + 1)
$796$  SUB2 = ZPUSH (SUB2 + 1)

       K = MEMORY(SUB2)                                             8

$797$  K = MEMORY(ABS SUB2)
$799$  K = MEMORY(ZPUSH SUB2)

       IF(K .EQ. '#') IER = 4                                    9  10

$554$  IF(MEMORY(SUB2) .EQ. '#') IER = 4
$800$  IF(ABS K .EQ. '#') IER = 4
$802$  IF(ZPUSH K .EQ. '#') IER = 4

 20    MEMORY(SUB1) = K                                            11

$559$  MEMORY(SUB1) = MEMORY(SUB2)
$803$  MEMORY(ABS SUB1) = K
$805$  MEMORY(ZPUSH SUB1) = K
$808$  MEMORY(SUB1) = ZPUSH K

       IF(IER .NE. 0) GOTO 9999                                 12  13

$745$  IF(IER .GT. 0) GOTO 9999
$876$  IF(IER .NE. 0) RETURN

       IF(DLEN .LE. MLEN) GOTO 9999                             14  15

$254$  IF(DLEN .LE. SLEN) GOTO 9999
$749$  IF(DLEN .EQ. MLEN) GOTO 9999
$809$  IF(ABS DLEN .LE. MLEN) GOTO 9999
$811$  IF(ZPUSH DLEN .LE. MLEN) GOTO 9999
$812$  IF(DLEN .LE. ABS MLEN) GOTO 9999
$814$  IF(DLEN .LE. ZPUSH MLEN) GOTO 9999
$878$  IF(DLEN .LE. MLEN) RETURN

       I = LOOPHI + 1                                              16

$815$  I = ABS LOOPHI + 1
$817$  I = ZPUSH LOOPHI + 1
$818$  I = ABS (LOOPHI + 1)
$820$  I = ZPUSH (LOOPHI + 1)

       LOOPHI = (DEST + DLEN) - 1                                  17

$821$  LOOPHI = (ABS DEST + DLEN) - 1
$823$  LOOPHI = (ZPUSH DEST + DLEN) - 1
$824$  LOOPHI = (DEST + ABS DLEN) - 1
```

```
$826$  LOOPHI = (DEST + ZPUSH DLEN) - 1
$827$  LOOPHI = ABS (DEST + DLEN) - 1
$829$  LOOPHI = ZPUSH (DEST + DLEN) - 1
$830$  LOOPHI = ABS ((DEST + DLEN) - 1)
$832$  LOOPHI = ZPUSH ((DEST + DLEN) - 1)

       DO 30 SUB1=I, LOOPHI                                        18

$833$  DO 30 SUB1=ABS I, LOOPHI
$835$  DO 30 SUB1=ZPUSH I, LOOPHI
$836$  DO 30 SUB1=I, ABS LOOPHI
$838$  DO 30 SUB1=I, ZPUSH LOOPHI
$891$  DO 9999 SUB1=I, LOOPHI
$893$  FOR 30 SUB1=I, LOOPHI

 30    MEMORY(SUB1) = ' '                                          19

$839$  MEMORY(ABS SUB1) = ' '
$841$  MEMORY(ZPUSH SUB1) = ' '

 9999  CONTINUE                                                    20

$883$  RETURN

       RETURN                                                      21
       END
MUTANT STATE FOR MOVENW

    FOR EXPERIMENT "MOVENW     " THIS IS RUN    7

    NUMBER OF TEST CASES =  11


    NUMBER OF MUTANTS =     893
    NUMBER OF DEAD MUTANTS =     821 ( 91.9%)
    NUMBER OF LIVE MUTANTS =       0 (  0.0%)
    NUMBER OF EQUIV MUTANTS =     72 (  8.1%)


    NUMBER OF MUTANTS WHICH DIED BY NON STANDARD MEANS   313
    NORMALIZED MUTANT RATIO 821.0%
    NUMBER OF MUTATABLE STATEMENTS =     21
    GIVING A MUTANTS/STATEMENT RATIO OF      42.52

    NUMBER OF DATA REFERENCES =     48
    NUMBER OF UNIQUE DATA REFERENCES =     16


   ALL MUTANT TYPES HAVE BEEN ENABLED


LISTING THE PROGRAM UNIT "MOVENM     " WITH SPECIFIED EQUIV MUTANTS

       SUBROUTINE MOVENM(SOURCE,SLEN,SDEC,DEST,DLEN,DDEC,TYPPE)
       LOGICAL NEGNO
```

```
       INTEGER X(5), PTNEGD, PTNEGS, K, SUB2, SUB1, LOOPHI, LEND
       INTEGER LENS, I, IHI, DDECPT, SDECPT, IER, STMT(3,10)
       INTEGER CODE(30), SYMTAB(10,9)
       CHAR MEMORY(425)
       INTEGER TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
       INPUT OUTPUT IER, MEMORY
       INPUT TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
       PTNEGS = (SOURCE + SLEN) - 1                              23

$4650$ PTNEGS = (ABS SOURCE + SLEN) - 1
$4652$ PTNEGS = (ZPUSH SOURCE + SLEN) - 1
$4653$ PTNEGS = (SOURCE + ABS SLEN) - 1
$4655$ PTNEGS = (SOURCE + ZPUSH SLEN) - 1
$4656$ PTNEGS = ABS (SOURCE + SLEN) - 1
$4658$ PTNEGS = ZPUSH (SOURCE + SLEN) - 1
$4659$ PTNEGS = ABS ((SOURCE + SLEN) - 1)
$4661$ PTNEGS = ZPUSH ((SOURCE + SLEN) - 1)

       PTNEGD = (DEST + DLEN) - 1                                24

$4662$ PTNEGD = (ABS DEST + DLEN) - 1
$4664$ PTNEGD = (ZPUSH DEST + DLEN) - 1
$4665$ PTNEGD = (DEST + ABS DLEN) - 1
$4667$ PTNEGD = (DEST + ZPUSH DLEN) - 1
$4668$ PTNEGD = ABS (DEST + DLEN) - 1
$4670$ PTNEGD = ZPUSH (DEST + DLEN) - 1
$4671$ PTNEGD = ABS ((DEST + DLEN) - 1)
$4673$ PTNEGD = ZPUSH ((DEST + DLEN) - 1)

       CALL UNPACK(MEMORY(PTNEGS),X,5)                           25

$4674$ CALL UNPACK(MEMORY(ABS PTNEGS),X,5)
$4676$ CALL UNPACK(MEMORY(ZPUSH PTNEGS),X,5)

       NEGNO = X(2) .EQ. '-'                                     26

$4545$ NEGNO = X(2) .GE. '-'
$4677$ NEGNO = ABS X(2) .EQ. '-'
$4679$ NEGNO = ZPUSH X(2) .EQ. '-'

       X(2) = ' '                                                27
       IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)                28 29

$4680$ IF(NEGNO) CALL PACK(X,MEMORY(ABS PTNEGS),5)
$4682$ IF(NEGNO) CALL PACK(X,MEMORY(ZPUSH PTNEGS),5)

       LENS = SLEN - SDEC                                        30

$4683$ LENS = ABS SLEN - SDEC
$4685$ LENS = ZPUSH SLEN - SDEC
$4686$ LENS = SLEN - ABS SDEC
$4689$ LENS = ABS (SLEN - SDEC)

       LEND = DLEN - DDEC                                        31

$4692$ LEND = ABS DLEN - DDEC
```

```
$4694$  LEND = ZPUSH DLEN - DDEC
$4695$  LEND = DLEN - ABS DDEC
$4698$  LEND = ABS (DLEN - DDEC)

        SDECPT = SOURCE + LENS                                   32

$4701$  SDECPT = ABS SOURCE + LENS
$4703$  SDECPT = ZPUSH SOURCE + LENS
$4704$  SDECPT = SOURCE + ABS LENS
$4707$  SDECPT = ABS (SOURCE + LENS)
$4709$  SDECPT = ZPUSH (SOURCE + LENS)

        DDECPT = DEST + LEND                                     33

$4710$  DDECPT = ABS DEST + LEND
$4712$  DDECPT = ZPUSH DEST + LEND
$4713$  DDECPT = DEST + ABS LEND
$4716$  DDECPT = ABS (DEST + LEND)
$4718$  DDECPT = ZPUSH (DEST + LEND)

        SUB1 = DDECPT - 1                                        34

$4719$  SUB1 = ABS DDECPT - 1
$4721$  SUB1 = ZPUSH DDECPT - 1
$4722$  SUB1 = ABS (DDECPT - 1)
$4724$  SUB1 = ZPUSH (DDECPT - 1)

        IF(SDEC .EQ. 0 .OR. DDEC .EQ. 0) GOTO 22              35  36

$4550$  IF(SDEC .LE. 0 .OR. DDEC .EQ. 0) GOTO 22
$4557$  IF(SDEC .EQ. 0 .OR. DDEC .LE. 0) GOTO 22

        IHI = (SDEC + SDECPT) - 1                                37

$4725$  IHI = (ABS SDEC + SDECPT) - 1
$4727$  IHI = (ZPUSH SDEC + SDECPT) - 1
$4728$  IHI = (SDEC + ABS SDECPT) - 1
$4730$  IHI = (SDEC + ZPUSH SDECPT) - 1
$4731$  IHI = ABS (SDEC + SDECPT) - 1
$4733$  IHI = ZPUSH (SDEC + SDECPT) - 1
$4734$  IHI = ABS ((SDEC + SDECPT) - 1)
$4736$  IHI = ZPUSH ((SDEC + SDECPT) - 1)

        IF(DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1          38  39

$4300$  IF(++ DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1
$4563$  IF(DDEC .LT. SDEC) IHI = (DDEC + SDECPT) - 1
$4737$  IF(ABS DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1
$4739$  IF(ZPUSH DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1
$4740$  IF(DDEC .LE. ABS SDEC) IHI = (DDEC + SDECPT) - 1
$4742$  IF(DDEC .LE. ZPUSH SDEC) IHI = (DDEC + SDECPT) - 1
$4743$  IF(DDEC .LE. SDEC) IHI = (ABS DDEC + SDECPT) - 1
$4745$  IF(DDEC .LE. SDEC) IHI = (ZPUSH DDEC + SDECPT) - 1
$4746$  IF(DDEC .LE. SDEC) IHI = (DDEC + ABS SDECPT) - 1
$4748$  IF(DDEC .LE. SDEC) IHI = (DDEC + ZPUSH SDECPT) - 1
        *4 MORE*
```

```
        DO 20 SUB2=SDECPT, IHI                                40

$4755$  DO 20 SUB2=ABS SDECPT, IHI
$4757$  DO 20 SUB2=ZPUSH SDECPT, IHI
$4758$  DO 20 SUB2=SDECPT, ABS IHI
$4760$  DO 20 SUB2=SDECPT, ZPUSH IHI
$5092$  FOR 20 SUB2=SDECPT, IHI

        SUB1 = SUB1 + 1                                       41

$4761$  SUB1 = ABS SUB1 + 1
$4763$  SUB1 = ZPUSH SUB1 + 1
$4764$  SUB1 = ABS (SUB1 + 1)
$4766$  SUB1 = ZPUSH (SUB1 + 1)

        K = MEMORY(SUB2)                                      42

$4767$  K = MEMORY(ABS SUB2)
$4769$  K = MEMORY(ZPUSH SUB2)

        IF(K .EQ. '#') IER = 4                             43 44

$2242$  IF(K .EQ. '#') IER = DLEN
$2244$  IF(K .EQ. '#') IER = LENS
$2245$  IF(K .EQ. '#') IER = SDEC
$2247$  IF(K .EQ. '#') IER = DDEC
$3467$  IF(MEMORY(SUB2) .EQ. '#') IER = 4
$4770$  IF(ABS K .EQ. '#') IER = 4
$4772$  IF(ZPUSH K .EQ. '#') IER = 4

 20     MEMORY(SUB1) = K                                      45

$3484$  MEMORY(SUB1) = MEMORY(SUB2)
$4773$  MEMORY(ABS SUB1) = K
$4775$  MEMORY(ZPUSH SUB1) = K
$4776$  MEMORY(SUB1) = ABS K
$4778$  MEMORY(SUB1) = ZPUSH K

        IF(IER .NE. 0) GOTO 50                             46 47

$4581$  IF(IER .GT. 0) GOTO 50
$5026$  IF(IER .NE. 0) GOTO 40

 22     IF(DDEC .LE. SDEC) GOTO 30                         48 49

$4779$  IF(ABS DDEC .LE. SDEC) GOTO 30
$4782$  IF(DDEC .LE. ABS SDEC) GOTO 30

        I = SUB1 + 1                                          50

$4785$  I = ABS SUB1 + 1
$4787$  I = ZPUSH SUB1 + 1
$4788$  I = ABS (SUB1 + 1)
$4790$  I = ZPUSH (SUB1 + 1)
```

```
       IHI = (DEST + DLEN) - 1                                    51

$4791$  IHI = (ABS DEST + DLEN) - 1
$4793$  IHI = (ZPUSH DEST + DLEN) - 1
$4794$  IHI = (DEST + ABS DLEN) - 1
$4796$  IHI = (DEST + ZPUSH DLEN) - 1
$4797$  IHI = ABS (DEST + DLEN) - 1
$4799$  IHI = ZPUSH (DEST + DLEN) - 1
$4800$  IHI = ABS ((DEST + DLEN) - 1)
$4802$  IHI = ZPUSH ((DEST + DLEN) - 1)

       DO 25 SUB1=I, IHI                                          52

$1168$  DO 25 SUB1=I, PTNEGD
$4803$  DO 25 SUB1=ABS I, IHI
$4805$  DO 25 SUB1=ZPUSH I, IHI
$4806$  DO 25 SUB1=I, ABS IHI
$4808$  DO 25 SUB1=I, ZPUSH IHI
$5073$  DO 30 SUB1=I, IHI
$5093$  FOR 25 SUB1=I, IHI

 25    MEMORY(SUB1) = '0'                                         53

$4809$  MEMORY(ABS SUB1) = '0'
$4811$  MEMORY(ZPUSH SUB1) = '0'

 30    LOOPHI = LEND                                              54

$4812$  LOOPHI = ABS LEND

       IF(LENS .LE. LEND) LOOPHI = LENS                        55 56

$1283$  IF(LENS .LE. LOOPHI) LOOPHI = LENS
$4359$  IF(++ LENS .LE. LEND) LOOPHI = LENS
$4591$  IF(LENS .LT. LEND) LOOPHI = LENS
$4815$  IF(ABS LENS .LE. LEND) LOOPHI = LENS
$4818$  IF(LENS .LE. ABS LEND) LOOPHI = LENS
$4821$  IF(LENS .LE. LEND) LOOPHI = ABS LENS

       SUB1 = DDECPT                                              57

$4824$  SUB1 = ABS DDECPT
$4826$  SUB1 = ZPUSH DDECPT

       SUB2 = SDECPT                                              58

$4827$  SUB2 = ABS SDECPT
$4829$  SUB2 = ZPUSH SDECPT

       IF(LEND .EQ. 0) GOTO 50                                 59 60

$2338$  IF(LEND .EQ. IER) GOTO 50
$4599$  IF(LEND .LE. 0) GOTO 50

       IF(LENS .EQ. 0) GOTO 41                                 61 62
```

```
$1443$  IF(LOOPHI .EQ. 0) GOTO 41
$4606$  IF(LENS .LE. 0) GOTO 41

        DO 40 I=1, LOOPHI                                              63

$1446$  DO 40 SOURCE=1, LOOPHI
$1447$  DO 40 SLEN=1, LOOPHI
$1450$  DO 40 DLEN=1, LOOPHI
$1453$  DO 40 SDEC=1, LOOPHI
$1455$  DO 40 DDEC=1, LOOPHI
$1456$  DO 40 SDECPT=1, LOOPHI
$1457$  DO 40 DDECPT=1, LOOPHI
$1459$  DO 40 IHI=1, LOOPHI
$1461$  DO 40 K=1, LOOPHI
$1463$  DO 40 LOOPHI=1, LOOPHI
        *4 MORE*

        SUB1 = SUB1 - 1                                                64

$4833$  SUB1 = ABS SUB1 - 1
$4835$  SUB1 = ZPUSH SUB1 - 1
$4836$  SUB1 = ABS (SUB1 - 1)
$4838$  SUB1 = ZPUSH (SUB1 - 1)

        SUB2 = SUB2 - 1                                                65

$4839$  SUB2 = ABS SUB2 - 1
$4841$  SUB2 = ZPUSH SUB2 - 1
$4842$  SUB2 = ABS (SUB2 - 1)
$4844$  SUB2 = ZPUSH (SUB2 - 1)

        K = MEMORY(SUB2)                                               66

$4845$  K = MEMORY(ABS SUB2)
$4847$  K = MEMORY(ZPUSH SUB2)

        IF(K .EQ. '#') IER = 4                                     67  68

$3670$  IF(MEMORY(SUB2) .EQ. '#') IER = 4
$4848$  IF(ABS K .EQ. '#') IER = 4
$4850$  IF(ZPUSH K .EQ. '#') IER = 4

 40     MEMORY(SUB1) = K                                               69

$3688$  MEMORY(SUB1) = MEMORY(SUB2)
$4851$  MEMORY(ABS SUB1) = K
$4853$  MEMORY(ZPUSH SUB1) = K
$4856$  MEMORY(SUB1) = ZPUSH K

        IF(IER .NE. 0) GOTO 50                                     70  71

$4623$  IF(IER .GT. 0) GOTO 50
$5050$  IF(IER .NE. 0) GOTO 20

        IF(LEND .LE. LENS) GOTO 50                                 72  73
```

```
$1743$  IF(LEND .LE. LOOPHI) GOTO 50
$4857$  IF(ABS LEND .LE. LENS) GOTO 50
$4859$  IF(ZPUSH LEND .LE. LENS) GOTO 50
$4860$  IF(LEND .LE. ABS LENS) GOTO 50
$4862$  IF(LEND .LE. ZPUSH LENS) GOTO 50

 41     IHI = SUB1 - 1                                          74

$4863$  IHI = ABS SUB1 - 1
$4865$  IHI = ZPUSH SUB1 - 1
$4866$  IHI = ABS (SUB1 - 1)
$4868$  IHI = ZPUSH (SUB1 - 1)

        DO 45 I=DEST, IHI                                       75

$4869$  DO 45 I=ABS DEST, IHI
$4871$  DO 45 I=ZPUSH DEST, IHI
$4872$  DO 45 I=DEST, ABS IHI
$4874$  DO 45 I=DEST, ZPUSH IHI
$5091$  DO 50 I=DEST, IHI
$5095$  FOR 45 I=DEST, IHI

 45     MEMORY(I) = '0'                                         76

$4875$  MEMORY(ABS I) = '0'
$4877$  MEMORY(ZPUSH I) = '0'

 50     X(2) = '-'                                              77
        IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)              78 79

$4878$  IF(NEGNO) CALL PACK(X,MEMORY(ABS PTNEGS),5)
$4880$  IF(NEGNO) CALL PACK(X,MEMORY(ZPUSH PTNEGS),5)

        IF(.NOT. (NEGNO .AND. TYPPE .EQ. 2)) RETURN          80 81

$4881$  IF(.NOT. (NEGNO .AND. ABS TYPPE .EQ. 2)) RETURN
$4883$  IF(.NOT. (NEGNO .AND. ZPUSH TYPPE .EQ. 2)) RETURN

        CALL UNPACK(MEMORY(PTNEGD),X,5)                         82

$57$    CALL UNPACK(MEMORY(PTNEGD),X,4)
$2560$  CALL UNPACK(MEMORY(PTNEGD),X,SDEC)
$2572$  CALL UNPACK(MEMORY(PTNEGD),X,TYPPE)
$3015$  CALL UNPACK(MEMORY(PTNEGD),X,1)
$3016$  CALL UNPACK(MEMORY(PTNEGD),X,2)
$4884$  CALL UNPACK(MEMORY(ABS PTNEGD),X,5)
$4886$  CALL UNPACK(MEMORY(ZPUSH PTNEGD),X,5)

        X(2) = '-'                                              83

$2593$  X(TYPPE) = '-'

        CALL PACK(X,MEMORY(PTNEGD),5)                           84

$4887$  CALL PACK(X,MEMORY(ABS PTNEGD),5)
$4889$  CALL PACK(X,MEMORY(ZPUSH PTNEGD),5)
```

```
      RETURN
      END
```

MUTANT ELIMINATION PROFILE FOR MOVENM

| MUTANT TYPE | TOTAL | DEAD | | LIVE | | EQUIV | |
|---|---|---|---|---|---|---|---|
| CONSTANT REPLACEMENT | 64 | 63 | 98.4% | 0 | 0.0% | 1 | 1.6% |
| SCALAR VARIABLE REPLACEME | 1920 | 1906 | 99.3% | 0 | 0.0% | 14 | 0.7% |
| SCALAR FOR CONSTANT REP. | 630 | 622 | 98.7% | 0 | 0.0% | 8 | 1.3% |
| CONSTANT FOR SCALAR REP. | 331 | 331 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| SOURCE CONSTANT REPLACEME | 102 | 100 | 98.0% | 0 | 0.0% | 2 | 2.0% |
| ARRAY REF. FOR CONSTANT R | 179 | 179 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| ARRAY REF. FOR SCALAR REP | 547 | 543 | 99.3% | 0 | 0.0% | 4 | 0.7% |
| COMPARABLE ARRAY NAME RE | 40 | 40 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| CONSTANT FOR ARRAY REF RE | 40 | 40 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| SCALAR FOR ARRAY REF REP. | 315 | 315 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| ARRAY REF. FOR ARRAY REF. | 75 | 75 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| UNARY OPERATOR INSERTION | 191 | 189 | 99.0% | 0 | 0.0% | 2 | 1.0% |
| ARITHMETIC OPERATOR REPLA | 107 | 107 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| RELATIONAL OPERATOR REPLA | 98 | 89 | 90.8% | 0 | 0.0% | 9 | 9.2% |
| LOGICAL CONNECTOR REPLACE | 10 | 10 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| ABSOLUTE VALUE INSERTION | 240 | 93 | 38.8% | 0 | 0.0% | 147 | 61.3% |
| STATEMENT ANALYSIS | 29 | 29 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| STATEMENT DELETION | 35 | 35 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| RETURN STATEMENT REPLACEM | 61 | 61 | 100.0% | 0 | 0.0% | 0 | 0.0% |
| GOTO STATEMENT REPLACEMEN | 49 | 47 | 95.9% | 0 | 0.0% | 2 | 4.1% |
| DO STATEMENT END REPLACEM | 32 | 25 | 78.1% | 0 | 0.0% | 7 | 21.9% |

MUTANT STATE FOR MOVENM

FOR EXPERIMENT "MOVENM " THIS IS RUN 22

NUMBER OF TEST CASES = 41

NUMBER OF MUTANTS = 5095
NUMBER OF DEAD MUTANTS = 4899 ( 96.2%)
NUMBER OF LIVE MUTANTS = 0 ( 0.0%)
NUMBER OF EQUIV MUTANTS = 196 ( 3.8%)

NUMBER OF MUTANTS WHICH DIED BY NON STANDARD MEANS 2206
NORMALIZED MUTANT RATIO *****%
NUMBER OF MUTATABLE STATEMENTS = 63
GIVING A MUTANTS/STATEMENT RATIO OF 80.87

NUMBER OF DATA REFERENCES = 158
NUMBER OF UNIQUE DATA REFERENCES = 32

ALL MUTANT TYPES HAVE BEEN ENABLED

## Appendix D

LISTING THE PROGRAM UNIT "MOVEED     "

```
      SUBROUTINE MOVEED(SOURCE,SLEN,SDEC,DEST,DLEN,PLEN,PDIG,PDEC,
    * PIC,IER)
      LOGICAL SUPRES, NEGNO
      INTEGER X(5), SUB2, SUB1, IHI, PLDIG, IVAR, I, SCOUNT, DESTHI
      INTEGER CHAR, PDIGLN, SDIG, SARRAY(50), PICST, DDEC
      INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
      CHAR MEMORY(310)
      INTEGER IER
      CHAR PIC(10)
      INTEGER PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      INPUT OUTPUT MEMORY, IER
      INPUT PIC, PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      SUPRES = .TRUE.                                          87
      DO 5 I=1, PLEN                                           88
5     SARRAY(I) = '0'                                          89
      PLDIG = PDIG - PDEC                                      90
      SDIG = SLEN - SDEC                                       91
      IF(SDEC .EQ. 0) GOTO 11                              92  93
      SUB1 = PLDIG                                             94
      SUB2 = (SOURCE + SDIG) - 1                               95
      DO 10 I=1, SDEC                                          96
      SUB1 = SUB1 + 1                                          97
      SUB2 = SUB2 + 1                                          98
      IF(MEMORY(SUB2) .EQ. '#') IER = 4                    99 100
10    SARRAY(SUB1) = MEMORY(SUB2)                             101
      IF(IER .NE. 0) GOTO 101                             102 103
11    IF(SDIG .EQ. 0 .OR. PLDIG .EQ. 0) GOTO 16           104 105
      IHI = PLDIG                                             106
      IF(SDIG .LT. PLDIG) IHI = SDIG                      107 108
      SUB1 = PLDIG + 1                                        109
      SUB2 = SOURCE + SDIG                                    110
      DO 15 I=1, IHI                                          111
      SUB1 = SUB1 - 1                                         112
      SUB2 = SUB2 - 1                                         113
      IF(MEMORY(SUB2) .EQ. '#') IER = 4                   114 115
15    SARRAY(SUB1) = MEMORY(SUB2)                             116
      IF(IER .NE. 0) GOTO 101                             117 118
16    SUB1 = (SOURCE + SLEN) - 1                              119
      CALL UNPACK(MEMORY(SUB1),X,2)                           120
      NEGNO = X(2) .EQ. '-'                                   121
      SUB1 = DEST                                             122
      SCOUNT = 0                                              123
      DO 100 I=1, PLEN                                        124
      SUB1 = SUB1 + 1                                         125
      IF(SUB1 .GT. DLEN + DEST) GOTO 101                  126 127
      CHAR = PIC(I)                                           128
      IF(PIC(I) .EQ. '9') SUPRES = .FALSE.                129 130
      IF(SARRAY(SCOUNT + 1) .NE. '0') SUPRES = .FALSE.    131 132
      IF(CHAR .NE. '-') GOTO 20                           133 134
      MEMCRY(SUB1 - 1) = ' '                                  135
      IF(NEGNO) MEMORY(SUB1 - 1) = '-'                    136 137
```

```
      IF(I .EQ. 1) GOTO 100                                            138 139
      SCOUNT = SCOUNT + 1                                                  140
      IF(.NOT. SUPRES) GOTO 99                                         141 142
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '             143 144
      GOTO 100                                                             145
20    IF(CHAR .NE. '+') GOTO 30                                        146 147
      IF(NEGNO) MEMORY(SUB1 - 1) = '-'                                 148 149
      IF(.NOT. NEGNO) MEMORY(SUB1 - 1) = '+'                           150 151
      IF(I .EQ. 1) GOTO 100                                            152 153
      SCOUNT = SCOUNT + 1                                                  154
      IF(.NOT. SUPRES) GOTO 99                                         155 156
      IF(MEMORY(SUB1 - 2) .EQ. '+') MEMORY(SUB1 - 2) = ' '             157 158
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '             159 160
      GOTO 100                                                             161
30    IF(CHAR .NE. '$') GOTO 40                                        162 163
      MEMORY(SUB1 - 1) = '$'                                               164
      IF(I .EQ. 1) GOTO 100                                            165 166
      SCOUNT = SCOUNT + 1                                                  167
      IF(.NOT. SUPRES) GOTO 99                                         168 169
      IF(MEMORY(SUB1 - 2) .EQ. '$') MEMORY(SUB1 - 2) = ' '             170 171
      GOTO 100                                                             172
40    IF(CHAR .NE. '*') GOTO 50                                        173 174
      SCOUNT = SCOUNT + 1                                                  175
      IF(.NOT. SUPRES) GOTO 99                                         176 177
      MEMORY(SUB1 - 1) = '*'                                               178
      GOTO 100                                                             179
50    IF(CHAR .NE. 'Z') GOTO 55                                        180 181
      SCOUNT = SCOUNT + 1                                                  182
      IF(.NOT. SUPRES) GOTO 99                                         183 184
      MEMORY(SUB1 - 1) = ' '                                               185
      GOTO 100                                                             186
55    IF(CHAR .NE. '9') GOTO 60                                        187 188
      SCOUNT = SCOUNT + 1                                                  189
      MEMORY(SUB1 - 1) = SARRAY(SCOUNT)                                    190
      GOTO 100                                                             191
60    IF(CHAR .NE. 'B') GOTO 70                                        192 193
      MEMORY(SUB1 - 1) = ' '                                               194
      GOTO 100                                                             195
70    IF(CHAR .NE. '/') GOTO 80                                        196 197
      MEMORY(SUB1 - 1) = '/'                                               198
      GOTO 100                                                             199
80    IF(CHAR .NE. 'V') GOTO 81                                        200 201
      SUB1 = SUB1 - 1                                                      202
      GOTO 100                                                             203
81    IF(CHAR .NE. '.') GOTO 82                                        204 205
      MEMORY(SUB1 - 1) = '.'                                               206
      GOTO 100                                                             207
82    IF(CHAR .NE. ',') GOTO 83                                        208 209
      IF(.NOT. SUPRES) MEMORY(SUB1 - 1) = ','                          210 211
      IF(SUPRES) MEMORY(SUB1 - 1) = ' '                                212 213
      GOTO 100                                                             214
83    IER = 3                                                              215
      GOTO 101                                                             216
99    MEMORY(SUB1 - 1) = SARRAY(SCOUNT)                                    217
100   CONTINUE                                                             218
101   RETURN                                                               219
```

END

# Bibliography

[Acree, 1979]
A.T. Acree, R.A. DeMillo, T.A. Budd, R.J. Lipton, and F.G. Sayward. "Mutation analysis." Technical Report GIT-ICS-79/08, Georgia Institute of Technology, 1979.

[Acree, 1980]
A. T. Acree, **On Mutation**, Ph.D. Thesis, Georgia Institute of Technology.

[Agarwal, 1979]
Vinod K. Agarwal and Gerald M. Masson, "Recursive Coverage Projection of Test Sets,", **IEEE Transactions on Computers**, Volume C-28(1):865-870, November 1979.

[Aho, 1975]
A. Aho and J. Ullman. **The Theory of Parsing Translation and Compiling, Vol 2: Compiling,** Prentice-Hall, 1975.

[Baldwin, 1979]
D. Baldwin and F. Sayward. "Heuristics for Determining Equivalence of Program Mutations," Yale University, Department of Computer Science Research Report, No. 276, 1979.

[Boyer, 1975]
R.S. Boyer, B. Elspas, and K.N. Levitt. "SELECT: A formal system for testing and debugging programs by symbolic executgion." **SIGPLAN Notices** 10(6):234-245, June 1975.

[Bradley, 1970]
G.H. Bradley. "Algorithm and bound for the greatest common divisor of n integers." **Communications of the ACM** 13 (7): 433-436, July 1970.

[Brooks, 1979]
Martin Brooks, **Autoatic Generation of Test Data for Recursive Programs having Simple Errors**, Ph.D. Thesis, Stanford University.

[Brown, 1975]
J.R. Brown and M. Lipow. "Testing for software reliability." **Proceedings of the 1975 International Conference on Reliable Software** (IEEE catalog number 75 CHO 940-7CSR), pages 518-527.

[Budd, 1978]
T.A. Budd and R.J. Lipton. "Mutation analysis of decision table programs." **Proceedings of the 1978 Conference on Information Sciences and Systems**, pages 346-349. The Johns Hopkins University, 1978.

[Budd, 1978a]
T.A. Budd and R.J. Lipton. "Proving LISP programs using test data." **Digest for the Workshop on Software Testing and Test Documentation**, pages 374-403, 1978.

[Budd, 1978b]
T.A. Budd, R.A. DeMillo, R.J. Lipton and F.G. Sayward. "The Design of a prototype mutation system for program testing," **Proc. 1978 NCC, AFIPS Conference Record**, pp. 623-627.

[Budd, 1980]
T.A. Budd. **Mutation analysis of program test data.** PhD thesis, Yale University.

[Budd, 1980a]
Timothy A. Budd and Dana Angluin, "Two Notions of Correctness and their Relation to Testing," Report TR 80-19, Department of Computer Science, University of Arizona.

[Budd, 1980b]
T.A. Budd, R.A. Demillo, R.J. Lipton and F.G. Sayward. " Theoretical and empirical studies of using program mutation to test the functional correctness of programs." **Proc. 1980 ACM Symposium on Principles of Programming Languages**, January, 1980, pp. 220-233.

[Budd, 1981]
Timothy A. Budd, "Mutation Analysis: Ideas, Examples, Problems, and Prospects," in **Computer Program Testing**, B. Chandrasekaran and S. Radicchi (eds.), North-Holland, pp. 129-148.

[Budd, 1982]
Timothy A. Budd, "A Portable Mutation System", manuscript, Department of Computer Science, University of Arizona.

[Burns, 1978]
J. Burns. "The stability of test data from program mutation," Digest for the Workshop on Software Testing and Test Documentation, Fort Lauderdale, Fla. 1978, pp. 324-334.

[Chang, 1970]
H.Y. Chang. **Fault diagnosis of digital systems.** Wiley- Interscience, 1970.

[Davis, 1958]
Martin Davis, **Computability and Unsolvability**, McGraw-Hill.

[DeMillo, 1978]
R.A. DeMillo and R.J. Lipton. "A probabilistic remark on algebraic program testing," **Information Processing Letters**, Vol. 7(4). (June, 1978), pp 193-195.

[DeMillo, 1978a]
R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Hints on test data selection: Help for the practicing programmer." **Computer** 11(4): 34-43, April 1978.

[DeMillo, 1979]
R.A. DeMillo, R.J. Lipton and A.J. Perlis. "Social Processes and proofs of theorems and programs," **CACM** Vol 22(5), (May, 1979), pp. 271-280.

[DeMillo, 1979a]
R.A. DeMillo, R.J. Lipton and F.G. Sayward, "Program mutation: A new approach to program testing," **INFOTECH State of the Art Report on Software Testing**, Vol. 2, INFOTECH/SRA, 1979, pp. 107-127 [Note: also see comentaries in Volume 1].

[Duran, 1982]
Joe W. Duran and Simeon C. Ntafos, "An Evaluation of Random Testing", manuscript, Department of Mathematical Sciences, University of Texas at Dallas.

[Foster, 1978]
K. Foster. "Error sensitive test cases." **Digest for the Workshop on Software Testing and Test Documentation**, pates 206-225, 1978.

[Gannon, 1983]
Carolyn Gannon, "Software Error Studies", **Proceedings** NSIA **National Conference on Software Test and Evaluation** pp. I-1 - I-7.

[Geller, 1978]
M. Geller. "Test data as an aid in proving program correctness." **Communications of the ACM** 21(5):368-375, May 1978.

[Gerhart, 1976]
S.L. Gerhart and L. Yelowitz. "Observations of failibility in applications of modern programming methodologies." **IEEE Transactions on Software Engineering** SE-2(3): 195-207, September 1976.

[Gilb, 1977]
T. Gilb. **Software Metrics**, Winthrop, 1977.

[Goodenough, 1975]
J.B. Goodenough and S.L. Gerhart. "Towards a theory of test data selection." **IEEE Transactions on Software Engineering SE-1** (2):156-173, June 1979.

[Goodenough, 1979]
J.B. Goodenough. "A survey of program testing issues." In P. Wegner, editor, **Research Directions in Software Technology**, pages 316-340, MIT Press, 1979.

[Gould, 1974]
J.D. Gould and P. Drongowski. "An exploratory study of computer program debugging." **Human Factors** 16(3):258-277, May 1974.

[Hamlet, 1977]
R.G. Hamlet. "Testing programs with the aid of a compiler." **IEEE Transactions Software Engineering**, Vol. SE-3 (4), (July 1977), pp.

[Hamlet, 1978]
R.G. Hamlet. "Critique of reliability theory". **Digest for the Workshop on Software Testing and Test Documentation**, pages 57-69, 1978.

[Hanks, 1980]
Jeanne M. Hanks, **Testing Cobol Programs by Mutation**, M.S. Thesis, Georgia Institute of Technology.

[Hardy, 1975]
S. Hardy. "Synthesis of LISP programs from examples." **Proceedings of the Fourth International Joint Conference on Artificial Intelligence**, pages 240-245. Held in Tbilisi, Georgia, USSR, 1975.

[Henderson, 1972]
P. Henderson and R. Snowden. "An experiment in structured programming". BIT 12:38-53, 1972.

[Hoare, 1961]
C.A.R. Hoare. "Algorithm 65: FIND." **CACM**, Vol. 4(1), (January, 1961), p. 321.

[Hoare, 1971]
C.A.R. Hoare. "Proof of a program: FIND." **Communications of the ACM** 14(1): 31-45, January 1971.

[Hopcroft, 1969]
J.E. Hopcroft and J.D. Ullman. **Formal Languages and Their Relation to Automata**. Addison-Wesley, 1969.

[Howden, 1975]
W.E. Howden. "Methodology for the generation of program test data." **IEEE Transactions on Commputers** c-24(5): 554-560, May 1975.

[Howden, 1976]
W.E. Howden. "Reliability of the path analysis testing strategy." **IEEE Transactions on Software Engineering** SE-2(3):208-214, September 1976.

[Howden, 1976a]
W. E. Howden, "Algebraic Program Testing", Technical Report, University of California, San Diego.

[Howden, 1978]
W.E. Howden. "An evaluatiion of the effectiveness of symbolic testing." **Software: Practice and Experience** 8:381-397, 1978.

[Howden, 1982]
W. E. Howden, "Weak Mutation Testing", **IEEE Transactions on Software Engineering**, Volume SE-8(4): 371-379, July, 1982.

[Huang, 1975]
J.C. Huang. "An approach to program testing." **Journal of the ACM** 7(3):113-128, September 1975.

[IBM, 1966]
International Business Machines. **System/360 Scientific Subroutine Package**. IBM Application Program H20-0205-3,1966.

[Kernhigan, 1978]
B.W. Kernhigan and P. Plauger. **The Elements of Programming Style.** McGraw-Hill, 1978 (Second Ed).

[Knuth, 1971]
D.E. Knuth. "An empirical study of fortran programs." **Software Practice and Experience**, Vol. 1(2), (1971), pp. 105-134.

[Ledgard, 1973]
H. Ledgard. "The case for structured programming." **BIT** 13:45-57, 1973.

[Linger, 1979]
R.C. Linger, H.D. Mills and B.I. Witt. **Structured Programming Theory and Practice**, Addison-Wesley, 1979.

[Lipton, 1978]
R.J. Lipton and F.G. Sayward. "The status of research on program mutation". **Digest for the Workshop on Software Testing and Test Documentation**, pages 355-378, 1978.

[Manna, 1974]
Z. Manna. **The Mathematical Theory of Computation**, McGraw-Hill, 1974.

[Minsky, 1967]
Marvin Minsky, **Computation: Finite and Infinite Machines**, Prentice-Hall.

[Montalbano, 1974]
M. Montalbano. **Decision Tables.** Science Research Associates, 1974.

[Nauer, 1969]
P. Nauer. "Programming by action clusters." **BIT** 9, 250-258, 1969.

[Osterweil, 1974]
L.J. Osterweil and L.D. Fosdick. "Data flow analysis as an aid in documentation, assertion generation, validation and error detection." University of Colorado, Department of Computer Science, Technical Report No. CU-CS-055-74, 1974.

[Osterweil, 1978]
L.J. Osterweil and L.D. Fosdick. " Experiences with DAVE -- A FORTRAN program analyzer." **Proceedings of the 1978 AFIP National Computer Conference**, pages 909-915, 1978.

[Ostrand, 1978]
T.J. Ostrand and E.J. Weyuker. " Remarks on the theory of test data selection." Digest for Workshop on Software Testing and Test Documentation, Fort Lauderdale, Fla, 1978, pp. 1-18.

[Pollack, 1971]
S.L. Pollack, H.T. Hicks, and W.J. Harrison. **Decision Tables: Theory and Practice.** John Wiley and Sons, 1971.

[Schaefer, 1973]
M. Schaefer. **A Mathematical Theory of Global Program Optimization**, Prentice-Hall, 1973.

[Shaw, 1975]
D.E. Shaw, W.K. Swartout, and C.C. Green. "Inferring LISP programs from examples." **Proceedings of the Fourth International Joint Conference on Artificial Intelligence**, pages 260-267. Held in Tbilisi, Georgia, USSR, 1975.

[Summers, 1975]
P.D. Summers. **Program Construction from Examples**. PhD thesis, Yale University, 1975.

[Tanaka, 1981]
Akihiko Tanaka, **Equivalence Testing for Fortran Mutation System using Data Flow Analysis**, M.S. Thesis, Georgia Institute of Technology.

[Thayer, 1978]
T.A. Thayer, M. Lipow, E.C. Nelson. **Software Reliability**, North-Holland, 1978.

[Thibodeau, 1978]
R. Thibodeau, "The State-of-the-Art in Software Error Data Collection." General Research Corporation, January, 1978.

[White, 1978]
L.J. White, E.I. Cohen, and B. Chandrasekaran. **A Domain Strategy for Computer Program Testing**. Technical Report OSU-CISRC-TR-78-4, Ohio State University, 1978.

[Wirth, 1968]
N. Wirth. "PL360: A programming language for the 360 computer." **Journal of the ACM** 15( ): 37-74, 1968.

[Youngs, 1974]
E.A. Youngs. "Human errors in programming," **International Journal of Man-Machine Studies**, Volume 6 (1974), pp. 361-376.