AD-A083 078 GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/G 9/2
THEORETICAL AND EMPIRICAL STUDIES ON USING PROGRAM MUTATION TO --ETC(U)
FEB 80 T A BUDD, R A DEMILLO, R J LIPTON N00014-79-C-0231
UNCLASSIFIED GIT-ICS-80/01 ARO-15950.5-A-EL NL

1 of 1
AD
A083078

END
DATE
FILMED
5 80
DTIC

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 10 15956.5-A-EL 18 ARO 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER 12
4. TITLE (and Subtitle) 6 THEORETICAL AND EMPIRICAL STUDIES ON USING PROGRAM MUTATION TO TEST THE FUNCTIONAL CORRECTNESS OF PROGRAMS	5. TYPE OF REPORT & PERIOD COVERED 9 Technical repl. 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Timothy A. Budd Frederick G. Sayward Richard A. DeMillo Richard J. Lipton	8. CONTRACT OR GRANT NUMBER(s) 15 N00014-79-C-0231, ✓ DAAG29-78-G-0121
9. PERFORMING ORGANIZATION NAME AND ADDRESS Georgia Institute of Technology Atlanta, Georgia 30332	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 14 GIT-ICS-80/01
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P. O. Box 12211 Research Triangle Park, NC 27709	12. REPORT DATE 11 Feb 80 13. NUMBER OF PAGES 14 12 17
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) LEVEL II	15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer programs
computer programming
computer program verification
computer program reliability

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A framework for studying the program mutation testing method from both theoretical and empirical viewpoints is presented.

410079

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

GIT-ICS 80/01

THEORETICAL AND EMPIRICAL STUDIES ON USING PROGRAM MUTATION TO TEST THE FUNCTIONAL CORRECTNESS OF PROGRAMS[*]

Timothy A. Budd[†]
Richard A. DeMillo[††]
Richard J. Lipton[†††]
Frederick G. Sayward[†]

February 1980

[*] This work was supported in part by Grant DAAG-29-78-G-0121 from ARO and AIRMICS and by Grant N00014-79-C-0231 from ONR.

[†] Yale University

[††] Georgia Institute of Technology

[†††] University of California, Berkeley

THEORETICAL AND EMPIRICAL STUDIES ON USING PROGRAM MUTATION TO TEST THE FUNCTIONAL CORRECTNESS OF PROGRAMS

Timothy A. Budd (Yale), Richard A. DeMillo (Georgia Tech),
Richard J. Lipton (Berkeley), and Frederick G. Sayward (Yale)

1. Introduction

In testing for program correctness, the standard approaches [11,13,21,22,23,24,34] have centered on finding data D, a finite subset of all possible inputs to program P, such that

1) if for all x in D, $P(x) = f(x)$,
   then $P^* = f$

where f is a partial recursive function that specifies the intended behavior of the program and P^* is the function actually computed by program P. A major stumbling block in such formalizations has been that the conclusion of (1) is so strong that, except for trivial classes of programs, (1) is bound to be formally undecidable [23].

There is an undeniable tendency among practitioners to consider program testing an ad hoc human technique: one creates test data that intuitively seems to capture some aspect of the program, observes the program in execution on it, and then draws conclusions on the program's correctness based on the observations. To augment this undisciplined strategy, techniques have been proposed that yield quantitative information on the degree to which a program has been tested. (See Goodenough [14] for a recent survey.) Thus the tester is given an inductive basis for confidence that (1) holds for the particular application. Paralleling the undecidability of deductive testing methods, the inductive methods all have had trivial examples of failure [14,18,22,23].

These deductive and inductive approaches have had a common theme: all have aimed at the strong conclusion of (1). Program mutation [1,7,9,27], on the other hand, is a testing technique that aims at drawing a weaker, yet quite realistic, conclusion of the following nature:

(2) if for all x in D, $P(x) = f(x)$,
    then $P^* = f$ OR P is "pathological."

To paraphrase,

3) if P is not pathological
   and $P(x) = f(x)$ for all x in D
   then $P^* = f$.

Below we will make precise what is meant by "P is pathological"; for now it suffices to say that P not pathological means that P was written by a competent programmer who had a good understanding of the task to be performed. Therefore if P does not realize f it is "close" to doing so. This underlying hypothesis of program mutation has become known as the *competent programmer hypothesis*: either $P^* = f$ or some program Q "close" to P has the property $Q^* = f$.

To be more specific, program mutation is a testing method that proposes the following version of correctness testing:

Given that P was written by a competent programmer, find test data D for which $P(D) = f(D)$ implies $P^* = f$.

Our method of developing D, assuming either P or some program close to P is correct, is by eliminating the alternatives. Let ϕ be the set of programs close to P. We restate the method as follows:

Find test data D such that:
 i) for all x in D $P(x) = f(x)$ and
ii) for all Q in ϕ
    either $Q^* = P^*$
    or for some x in D, $Q(x) \ne P(x)$.

If test data D can be developed having properties (i) and (ii), then we say that D *differentiates* P from ϕ, alternatively P passes the ϕ mutant test.

The goal of this paper is to study, from both theoretical and experimental viewpoints, two basic questions:

Question 1: If P is written by a competent programmer and if P passes the ϕ mutant test with test data D, does $P^* = f$?

Note that, after formally defining ϕ for P in a fixed programming language L, an affirmative answer to question 1 reduces to showing that the competent programmer hypothesis holds for this L and ϕ.

We have observed that under many natural definitions of ϕ there is often a strong coupling

This work was supported in part by grant DAAG-29-78-G-0121 from ARO and AIRMICS and by grant N00014-79-C-0231 from ONR.

between members of ϕ and a small subset μ. That is, often one can reduce the problem of finding test data that differentiates P from ϕ to that of finding test data that differentiates P from μ. We will call this subset μ the *mutants* of P and the second question we will study involves the so-called *coupling effect* [9]:

Question 2 (Coupling Effect): If P passes the μ mutant test with data D, does P pass the ϕ mutant test with data D?

Intuitively, one can think of μ as representing the programs that are "very close" to P.

In the next section we will present two types of theoretical results concerning the two questions above: general results expressed in terms of properties of the language class L, and specific results for a class of decision table programs and for a subset of LISP. Portions of the work on decision tables and LISP have appeared elsewhere [5,6], but the presentations given here are both simpler and more unified. In the final section we present a system for applying program mutation to FORTRAN and we introduce a new type of software experiment, called a "beat the system" experiment, for evaluating how well our system approximates an affirmative response to the program mutation questions.

2. Theoretical Studies

Our major interest in studying program mutation from a theoretical viewpoint is to gain insight on where and how to apply the method to real programming languages. There are two possible study approaches: (i) For fixed L 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 captures the notion of "closeness." (That is, reflect the errors a competent programmer could have made in producing P.) Here ϕ is a subset of L. (ii) Simply let $\phi = L$.

For these initial studies we choose the simpler approach, (ii), in order to investigate the questions in terms of properties of L. Immediately we have the following:

Theorem 1: If there is an automatic method to generate data D that satisfies the mutation test for P, then the equivalence of P and any program Q in ϕ must be decidable.

The proof is trivial since we used the equivalence property in defining what is meant by "data that satisfies the mutation test." Furthermore, if we have such data, then to decide equivalence we merely execute the two programs. If the results agree, they are equivalent; if not, they obviously are not equivalent.

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 the sequel, however, the implication is that we must carefully choose the set ϕ to capture some very special properties of being "close" to the original program P.

For the remainder of this section we will consider two specific examples of program types.

2.1 Decision Tables

A "decision table" is a highly structured way of describing decision alternatives. Such tables are chiefly used in business and data processing applications [28,31], although they can also be used to organize test data selection predicates [13].

To form a decision table we have a set of conditions, a set of actions, and a table composed of two parts. Entries in the upper part are 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.

	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 the program on some 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 complete. We say it is consistent if there is at most one rule. We will assume that the program under test is consistent. We can also assume it is complete, since an incomplete decision table can always be turned into a complete one by adding further actions that merely return an error flag and further rules that are satisfied by the previously unmatched inputs.

We will also assume that no two rules specify exactly the same set of actions. We can do this with little loss of generality since two such rules can be combined with at most the addition of one new condition.

Given a decision table program P, let ϕ be the set of all consistent programs having the same conditions and actions as P. This means that members of ϕ differ from P only in the table portions or the number of rules they contain.

The mutants (μ) of P will be those members of ϕ that are formed by taking a *single* * entry and changing it first into a Y and then into an N entry. If P is consistent then all the mutants will be consistent. Some of these mutants may be equivalent to P. The mutant that changes position j in rule i from a * to a Y can be equivalent to P only if it is impossible for any input to satisfy rule i and not satisfy this condition.

There are at most two mutants for every table entry in P. This means there are no more than 2nm 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, an adequate mutation set need have only at most 2nm inputs.

We will make the following assumptions apply to all of ϕ:

1. The program P is both consistent and complete.
2. Given an example of input/output behavior, we can determine which rule was applied to produce the output from the input. In particular this implies that no two rules specify exactly the same set of actions.
3. There exists at least one input that satisfies each rule.
4. We can decide the equivalence of P and any member of μ.

We will investigate the power of a set of inputs that differentiates P from μ; in particular, we will show that this set in fact differentiates P from ϕ. Assume we have such a set D. We will assume that every rule in P is exercised at least once by some member of D, adding points if necessary to meet this condition. We can initially fail to meet this condition only if there are some rules that do not contain *s. We will note without further comment that we could have guaranteed this condition with mutants if we also mutated the action matrix, as was done in the original paper [5]. It now seems that to do so causes an unnecessary increase in the complexity of the proof for such a small matter.

Given any Q in ϕ, if for each x in D P(x) = Q(x) then we will say Q *tests equal* to P. Since each rule in P has a unique set of actions, by a simple counting argument we know 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 2: If D differentiates P from μ and Q tests equal to P, then for each rule in P the set of inputs satisfying the corresponding rule in Q is strictly larger than that of P.

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. If this were so no data that satisfied the rule in P could satisfy the rule in Q.

Now consider each * entry in P. There are two cases. If the change that replaces this * by a Y (the same argument holds for N) is equivalent, this means 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 of inputs accepted by Q.

On the other hand if this change is not equivalent, D contains points that while satisfying the rule 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 in P has a Y (or N) and the corresponding position in Q has a *. This strictly increases the size of the set of inputs accepted by this rule, giving our result. □

Theorem 3: If D differentiates P from μ then D differentiates P from ϕ.

Proof: Let P_i be the set of inputs accepted by rule i in P. Since P is consistent, the P_i are disjoint. Since P is complete, they cover the entire space of inputs. Each rule in Q must accept at least the set accepted by the corresponding rule in P. Since Q is consistent, it can satisfy no more. □

Recall that theorem 1 stated that we could form an adequate mutation set 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 the conditions are independent and therefore none of the mutants are equivalent. We can easily find examples where this is not true. This is the case whenever we have two condiitions where the question of whether the first condition implies the second is undecidable.

condition 1	Y
condition 2	*

Figure 2: Example where equivalence is undecidable

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 classic undecidable question [20] 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 3 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 sufficiant to eliminate all the mutants of program 1.

Program 1					Program 2			
N	Y	N	Y		*	*	*	*
*	*	*	*		N	Y	N	Y
Y	N	N	Y		*	*	*	*
*	*	*	*		Y	N	N	Y
X	X	O	O		X	X	O	O
O	O	X	X		O	O	X	X

Figure 3: A case not covered by the mutation test

We do not know 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 2^n possibilities.

2.2 LISP Programs

In this section we will consider programs written in the subset of LISP containing the functions CAR, CDR, and CONS and the predicate ATOM. A similar class of programs has been studied previously [17,32,33].

We will make the convention that all S-expressions (we will use the less clumsy locution *points*) have unique atoms. Certainly if two programs agree on all such points they must agree on all inputs; hence we do this without loss of generality.

We will call a LISP program a *selector* program if it is composed of just CAR and CDR. We will inductively define a *straight-line* program as a selector program or a program formed by the CONS of two other straight-line programs.

2.2.1 Straight-line programs

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

Theorem 4: If two selector programs return identical values 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 infer the subtree that was selected. □

We will say that a straight-line program P(X) is *well formed* 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 the definition should be clear: a program is well formed if it is not doing any more work than it needs to. Notice that being well formed is an observable property of programs, *independent of testing*.

We can define a measure of the complexity of straight-line programs by their CONS-depth, where CONS-depth is defined as follows:

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 5: If two well formed selector programs compute identically on any point for which they are both defined, then they must have the same CONS-depth.

Proof: Assume we have two programs P1 and P2 and a point X such that P1(X) = P2(X), yet the CONS-depth(P1) < CONS-depth(P2). This implies that there is at least one subtree in the structure of P2 that was produced by CONSing two straight-line programs while the same subtree in P1(X) was produced by a selector. But then the objects P2 CONSed must have an immediate ancestor in X, contradicting the fact that P2 is well formed. □

Theorem 6: If two well formed straight-line programs agree on any point X for which they are both defined, then they must agree on all points.

Proof: The proof will be by induction on the CONS-depth. By theorem 5 any two programs that agree on X must have the same CONS-depth. By theorem 4 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 P1 has CONS-depth n+1 then it must be of the form CONS(P11,P12) where P11 and P12 have CONS-depth no greater than n. Assume we have two programs P1 and P2 in this fashion. Then for all Y:

P1(Y) = P2(Y) IFF
CONS(P11(Y),P12(Y)) = CONS(P21(Y),P22(Y)) IFF
P11(Y) = P21(Y) and P12(Y) = P22(Y)

Hence by the induction hypothesis P1 and P2 must agree for all Y. □

We can easily generalize theorem 6 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 one can verify that if theorem 6 did not hold in the case of multiple arguments, one could construct two programs with single arguments for which it did not hold, 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.

2.2.2 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 *selector variables*, denoted $x_1,\dots,x_m$, and *constructor variables*, denoted $y_1,\dots,y_p$. A *program* will consist of a program body and a recurser. A *program body* consists of n statements, each statement composed of a *predicate* of the form ATOM($t(x_1)$) where t is a selector function and x_1 a selector variable, and a straight-line *output function* over the selector and constructor variables. A *recurser* is divided into two parts. The *constructor* part is composed of p assignment statements for each of the p constructor variables where y_i is assigned a straight-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 is evaluated in turn. If any predicate is undefined, so is the result 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.

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

```
Program $P(x_1,\dots,x_m,y_1,\dots,y_p)$ =
IF $p_1(x_{i1})$ THEN $f_1(x_1,\dots,x_m,y_1,\dots,y_p)$
ELSE IF $p_2(x_{i2})$ THEN $f_2(x_1,\dots,x_m,y_1,\dots,y_p)$
...
ELSE IF $p_n(x_{in})$ THEN $f_n(x_1,\dots,x_m,y_1,\dots,y_p)$
ELSE
$y_1 := g_1(y_1,x_1,\dots,x_m)$
...
$y_p := g_p(y_p,x_1,\dots,x_m)$
$x_1 := n_1(x_1)$
...
$x_m := n_m(x_m)$
$P(x_1,\dots,x_m,y_1,\dots,y_p)$
```

Figure 4: An example recursive *program*

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 $\ell \geq 0$ there exists at least one input that causes the *program* to recurse ℓ times before exiting with output function k.

Let ϕ 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 D that differentiates P from all members of ϕ. The set of mutants μ will be described in the course of the proof, as they enter into the arguments. The proof will proceed in several smaller steps:

In subsection 1 we give some basic definitions and demonstrate some tools that we will use in later sections. Subsection 2 shows how to use testing to bound the depth of the selector functions. In subsection 3 we narrow the form of the selector functions still further, and finally in subsection 4 show that they must exactly match P. In subsection 5 we deal with the points tested by the predicates, and in subsection 6 we give the main theorem. Subsection 7 concludes with some comments on the difficulty of proving a program correct in this manner and ways in which the results here could be strengthened.

2.2.3 Definitions and tools

We will use capitol letters from the end of the alphabet (X, Y, and Z) to represent vectors of inputs. Hence we will refer to P(X) rather than $P(x_1,\dots,x_m,y_1,\dots,y_p)$. 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 *positions* 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 *depth* 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 *size* of an input X will be the maximum depth of any of the atoms in X.

We can extend the definition of $\leq$ to the space of inputs by saying $X \leq Y$ 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 *touched*. Call the n positions touched by the predicates of P without going into recursion the *primary positions* 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 7: Given $\ell \geq 0$ and $1 \leq i \leq n$ we can construct an input X so that *P(X)* is *defined and when given* X as an input P recurses ℓ times before exiting by output function i.

Proof: 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 ℓ times and reach the predicate i. Then mark in RED the point touched by predicate i after recursing ℓ 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 ℓ times, and each position used by output function i after recursing ℓ 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 7 allows one to find the *smallest* X such that the indicated conditions hold. If a is the position in question, call this point the *minimal a* *point*. Freedom 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 6 F_X is defined by this single point.

Theorem 8: 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 ℓ and i be the constants such that on input X, P recurses ℓ 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 $\ell+1$ times is an interior position in X, but since X is bounded there must be a smallest $k > \ell$ such that the predicate $p_i(R(x_j))$ is either true or undefined. Using theorem 7 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

$$f_i(R^{\ell}(X),R^{\ell}(Y)) \neq f_i(R^{k}(X),C^{k}(Y))$$

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. The same argument as before goes through with the exception 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 9: 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 ϕ, 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 8 we can construct an input X such that P(X) is defined, $X \geq Z$, and $F_X \neq F_Z$. Let Y be X pruned at a.

We first assert 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 given input Y a program Q recurses ℓ 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 ℓ times and exit by output function i. But this implies by theorem 6 that $F_X = F_Y$, a contradiction. □

2.2.4 Bounding the depth of the recursion and predicate functions

Our first set of test inputs uses the procedure given in theorem 9 to demonstrate that each of the n primary positions in P are indeed touched.

Next, for *each* selector variable, use the procedure given in theorem 9 to show that the first n+1 positions (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 10: If Q is a program in ϕ that correctly processes these 2m(n+1) points, then the recursion selectors of Q have depth d or less.

Proof: Study 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 11: If $Q \in \phi$ 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 9. Let ℓ 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 8) 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 9 need be any larger than 1+2w+d. This fact will be of use in proving theorem 14.

2.2.5 Narrowing the form of the recursion selectors

We will say a *selector function f factors* 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 *simple factor* of g if f factors g and no function factors f other than f itself.

Let us denote by s_i i = 1,...,m the simple factors of r_i, the recursion selector functions. That is, for each variable i there is a constant ℓ_i so that the recursion selector r_i is s_i composed with itself ℓ_i times. Let q be the greatest common divisor of all the ℓs. 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 9, generate two points that demonstrate that position a must be touched. Let D_0 be the set containing all the $(2n + 2m(n+1) + 2m)$ points computed so far.

Theorem 12: If $Q \in \phi$ computes correctly on D_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 10 this is impossible. □

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

Theorem 13: If $Q \in \phi$ computes correctly on D_0 then there exists a constant r such that the recursion selectors of Q are exactly S^r.

Proof: We know by theorem 12 that 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 13 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 is gaining 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 9 gave us a method to demonstrate a position is touched. We now give the opposite: a way to demonstrate a position is not touched.

Theorem 14: If $Q \in \phi$ 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 ℓ 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 8 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. □

2.2.6 Recursion selectors must be the same as P

If $Q \in \phi$ executes correctly on D_0, then from theorem 13 we know the recursion selectors of Q must be S for some constant r. From theorem 10 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 13, 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 14) 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 D_1.

Theorem 15: If $Q \in \phi$ executes correctly on D_0 and D_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. □

2.2.7 Testing the primary positions of P

Consider each primary position separately. Assume that in some program Q in ϕ the position is not primary, but that it is touched after having recursed ℓ times. Let b be the position of a relative to $S^{q\ell}$. 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 14, 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 carefully 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 $n(d/(\text{depth of } S^q))$ mutant programs, hence at most twice this number of test points. Call this test set D_2.

Theorem 16: If $Q \in \phi$ executes correctly on D_0, D_1, and D_2 then the primary positions of Q are exactly those of P.

Notice that by theorem 6 this also gives us the following.

Theorem 17: The output functions of Q are exactly those of P.

2.2.8 Main theorem

Once we have the other elements fixed, the constructors are almost given to us. Remember one of the assumptions made in the beginning was 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 D_3. Using theorem 6 we then have

Theorem 18: 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 ϕ) is to try all possibilities. There are n! of these, some of which may be equivalent to the original program. Let D_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 19: Given a program P in ϕ, if $Q \epsilon \phi$ executes correctly on the test points constructed in theorems 10, 15, 16, and 18, then Q must be equivalent to P.

Corollary: Either P is correct or no program in ϕ realizes the intended function.

Corollary: If the competent programmer hypothesis holds then P is correct.

2.3 Discussion

We note that although the class of programs studied here is small, it is not vacuous. Several of the examples studied previously [17,32,33] can be expressed in our form.

We point out that even with the assumed bound on the depth of the output functions, we did not bound the number of CONS functions they can contain; hence there are an infinite number of programs in the set ϕ. This is true even after we have bounded the depth of the recursion selectors and the predicate selectors in theorem 11.

The most important aspect of this result is not the proof (which in fact has rather limited applicability) but the method of the proof. Once we have fixed the recursion selectors via test set D_0, the remainder of the arguments can be proved by constructing a small set of alternative programs (the 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 ϕ.

We note that although the proof of the result given here is rather long and tedious, the result is a procedure for proving correctness that is entirely mechanical. The user of such a procedure need have no knowledge of the proof that was used to validate the method, much as the user of a timesharing system need have no knowledge of how the operating system is implemented. This is the direction we feel research in testing should follow: finding mechanical methods that may be difficult to verify, but that once verified give an easy procedure for finding good test data.

3. Empirical Studies

A program mutation system, called EXPER [27], has been implemented to test ANSI FORTRAN programs. In building real testing tools for real programming languages, the issues that must be considered are:

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

Note that these two issues should trade off somewhat as time and space trade off in algorithms. But in addressing the first issue, the system must at least be tractable.

Conceivably, given a FORTRAN program P having N statements one could construct a mutant set μ having size exponential in N, each mutant being a reasonable alternative to P. What is done in EXPER, however, is to define μ via a set of 23 *mutant operators* that, upon analysis, result in μ's having size bounded roughly by the product of the number of data references (constants, scalar variables, and array references) times the number of unique data references. Mutant operators are very simple syntactic and semantic program transformation rules that act on P in only a local way. For example, one mutant operator changes a *single* occurrence of a binary operator in P to a syntactically legal alternative operator in forming a mutant identical to P in all but one symbol. Another mutant operator changes the semantics of a single DO loop to be interpreted as a FOR loop. Here the mutant is syntactically identical to P but, unlike P, precisely one of the mutant's k loop bodies might never be executed in spite of the controlling DO statement being executed. A description of the exact nature of the other mutant operators has already been published [7].

Although the complexity of the mutation system is now reasonable, one should question the effectiveness of applying program mutation with only simple alternatives since the remaining more complicated (but reasonable) alternatives are apparently overlooked. The coupling effect mentioned in section 1 indirectly addresses the more complicated alternatives of P: test data that causes all simple mutations of P to fail is so sensitive that it implicitly causes all complex combinations of them to fail.

We will illustrate a representative case of coupling in a FORTRAN program. The program is adapted from the IBM scientific subroutines package [25], a collection of statistical and scientific programs in fairly common use. The error was artificially inserted in a study by Gould and Drongowski [15]. The error occurs in the line that reads

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

but should read

```
      40 INN = UBO(2)
      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(I) = 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
      INTX = INN - 1
      DO 60 I=1, INTX
      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
      GOTO 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
      GOTO 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
```

There are a number of mutants that discover this error. Consider, for example, the one that changes the statement

```
     IF (A(IJ) - TEMP) 75,65,65
to
     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 can 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, this is out of bounds, and the error is discovered.

We shall not directly address the coupling effect further here -- evidence for it has been previously reported in many sources [7,8,9,27,30] -- but instead will report on experiments aimed at evaluating issue (2) above.

The ultimate evaluation of any program testing system involves examining the following question:

Are there incorrect programs that pass the system's test?

Since, as was argued above, the answer will always be yes for any system that tests real programs, a more interesting question is:

What types of errors are always detected by the system, *and what error types might be overlooked?*

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

The ideal experiment for evaluating a program mutation system would be the classic double-blind experiment. The experimenter 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. Classical statistical techniques are then used to evaluate the results. Unfortunately, the high cost of performing such controlled N-subject experiments makes them unfeasible.

We have, however, designed and performed a single-subject experiment, which we claim gives significant results in evaluating the FORTRAN mutation system. We call such an experiment a *beat the system* experiment. The single subject is someone having a very high level of programming expertise and much familiarity with the concepts of programming mutation in general and the FORTRAN mutation system in particular. The M programs now have one or more errors, and furthermore the subject *now has complete knowledge* 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 we have high confidence that these error types would 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 more investigation of mutant operators is needed -- the system might be weak *in detecting those error types.*

The beat the system experiment is an example of worst-case analysis, in that we attempt to find out how the system will perform under the worst possible circumstances. We note that the beat the system experiment is an extension of the reliability studies done previously [16,23]. These earlier studies, however, were directed at comparing two or more competing methodologies and deriving statistical information of the form "On the following samples of programs, method A discovered X% of the errors and method B discovered Y%." In the beat the system experiments we are much less concerned with the number of errors caught and much more concerned with the type of errors missed. Furthermore this information is not used to compare two methods but is designed to evaluate the mutation analysis system (EXPER) and to direct the search for new mutant operators that will improve the system.

For example, several of the programs we 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, forgetting that no matter what the limits are a DO statement will always (perhaps erroneously) loop at least once. *The way we chose to detect these errors was to* introduce a mutant that changed a DO statement into a FOR statement, bringing this fact to the programmer's attention and forcing him to derive data that indicated he had knowledge of this potential pitfall.

So far we have conducted beat the system *experiments on 11 programs,* all of which have been previously studied in the testing literature. (We wish to express our gratitude to Robert Hess, who was the subject in most of these experiments.) The appendix contains the appropriate references and further details on the programs and their errors.

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. 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. Goodenough and Gerhart [13] and Howden [22], among others, have attempted to construct a generally applicable system. Neither of their systems, to our minds, gives a sufficiently intuitive picture of the errors in any particular class. Therefore we have chosen to group the errors in these eleven programs 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.

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

The 25 errors in these 11 programs range from simple to extremely 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. We merely cannot guarantee their discovery. Table 5 gives the number of errors detected by error type. Of these 25 errors, only 8 would be caught using branch analysis.

	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

Figure 5: Number of errors detected by error type

One can notice that in three of these categories the errors are caused by the lack of certain constructs in the program. Since the testing method is being 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 EXPER system to detect these 5 errors is really not an indication of a weakness in the method per se; 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. We are constantly looking for patterns in the types of errors overlooked in order to discover new mutant operators that would aid in the detection of these errors.

4. Concluding Remarks

A framework for studying the program mutation testing method from both theoretical and empirical viewpoints has been presented. The initial results indicate that program testing, when studied under assumptions in addition to the standard "P is correct on test data D," is a fertile area for both theoretical and empirical research.

We remark that there is a very real trade-off in the two types of research we have presented. We can prove theorems that allow perfect testing of programs for very limited languages, languages that programmers would find rather unnatural. But undecidability stands in the way of generalizing these to real programming languages. Given this, we have gathered empirical evidence that implies that the type of results we would like to prove do indeed hold in the real world. Both types of results are useful and important in understanding the nature of program testing.

Although our specific results have dealt solely with program testing, we feel that the potential for developing other software methodologies that try to exploit some facet of the programming process, as illustrated by our use of the competent programmer hypothesis, should not be overlooked.

Finally, we feel that the type of experiments comparable to our beat the system approach are an example of experimental computer science adaptable to many other testing methodologies.

References

1] 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.

2] R. S. Boyer, B. Elspas, and K. N. Levitt. "SELECT: A formal system for testing and debugging programs by symbolic execution." *SIGPLAN Notices* 10(6):234-245, June 1975.

3] Gordon H. Bradley. "Algorithm and bound for the greatest common divisor of n integers." *Communications of the ACM* 13(7):433-436, July 1970.

4] 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. IEEE, 1975.

5] Timothy A. Budd and Richard 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.

6] Timothy A. Budd and Richard J. Lipton. "Proving LISP programs using test data." *Digest for the Workshop on Software Testing and Test Documentation*, pages 374-403, 1978.

7] Timothy A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, in preparation.

8] H. Y. Chang. *Fault Diagnosis of Digital Systems*. Wiley-Interscience, 1970.

9] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Hints on test data selection: Help for the practicing programmer." *Computer* 11(4):34-43, April 1978.

10] K. Foster. "Error sensitive test cases." *Digest for the Workshop on Software Testing and Test Documentation*, pages 206-225, 1978.

11] M. Geller. "Test data as an aid in proving program correctness." *Communications of the ACM* 21(5):368-375, May 1978.

12] Susan L. Gerhart and Lawrence Yelowitz. "Observations of fallibility in applications of modern programming methodologies." *IEEE Transactions on Software Engineering* SE-2(3): 195-207, September 1976.

13] John B. Goodenough and S. L. Gerhart.

"Towards a theory of test data selection." *IEEE Transactions on Software Engineering* SE-1 (2):156-173, June 1975.

[14] J. Goodenough. "A survey of program testing issues." In P. Wegner, editor, *Research Directions in Software Technology*, pages 316-340. MIT Press, 1979.

[15] John D. Gould and Paul Drongowski. "An exploratory study of computer program debugging." *Human Factors* 16(3):258-277, May 1974.

[16] Richard Hamlet. "Critique of reliability theory. *Digest for the Workshop on Software Testing and Test Documentation*, pages 57-69, 1978.

[17] Steven 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.

[18] P. Henderson and R. Snowden. "An experiment in structured programming." *BIT* 12:38-53, 1972.

[19] C. A. R. Hoare. "Proof of a program: FIND." *Communications of the ACM* 14(1):31-45, January 1971.

[20] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

[21] William E. Howden. "Methodology for the generation of program test data." *IEEE Transactions on Computers* C-24(5):554-560, May 1975.

[22] William E. Howden. "An evaluation of the effectiveness of symbolic testing." *Software: Practice and Experience* 8:381-397, 1978.

[23] William E. Howden. "Reliability of the path analysis testing strategy." *IEEE Transactions on Software Engineering* SE-2(3):208-214, September 1976.

[24] J. C. Huang. "An approach to program testing." *Journal of the ACM* 7(3):113-128, September 1975.

[25] International Business Machines. *System/360 Scientific Subroutine Package*. IBM Application Program H20-0205-3, 1966.

[26] H. Ledgart. "The case for structured programming." *BIT* 13:45-57, 1973.

[27] 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.

[28] M. Montalbano. *Decision Tables*. Science Research Associates, 1974.

[29] P. Naur. "Programming by action clusters." *BIT* 9:250-258, 1969.

[30] L. J. Osterweil and L. D. Fosdick. "Experiences with DAVE -- A FORTRAN program analyzer." *Proceedings of the 1976 AFIP National Computer Conference*, pages 909-915, 1976.

[31] S. L. Pollack, H. T. Hicks, and W. J. Harrison. *Decision Tables: Theory and Practice*. John Wiley and Sons, 1971.

[32] 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.

[33] Philip Dale Summers. *Program Construction from Examples*. PhD thesis, Yale University, 1975.

[34] 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.

[35] N. Wirth. "PL360: A programming language for the 360 computer." *Journal of the ACM* 15( ): 37-74, 1968.

Appendix

This appendix describes the 11 programs studied in the "beat the system" experiments.

The first program is written in an ALGOL dialect and initially appeared in a paper by Henderson and Snowden [18]. 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 Ledgart [26] and Gerhart and Yelowitz [[illegible]]. 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 on the EXPER system 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 the zero push operator in EXPER. If we had such an operator, an attempt to force the expression FIRST(input) to blank would certainly reveal the error.

The second program appears in a paper by Wirth describing the language PL-360 [35]. It is intended to take a vector of N numbers and sort them into decreasing order. It was also studied by Gerhart and Yelowitz [12]. 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.

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 [25]. It was analyzed and three artificial errors were inserted in a study by Gould and Drongowski [15]. As in the study by Howden [22] 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.

The fourth program appeared in an article by Geller in the *Communications of the ACM* [11]. 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" [9], 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.

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* [3]. 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 [25]. In each program three errors were artificially inserted in a study conducted by Gould and Drongowski [15].

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.

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.

The third program computes correlation coefficients. Two of the errors are detected with branch analysis; all three are detected with mutation analysis.

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 [4]. Lipton and Sayward [27] 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.

The tenth program is the FIND program from an article by C. A. R. Hoare [19]. The bug has been studied by the group developing the SELECT symbolic execution system [2]. 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 [9].

The last program, also written in ALGOL, appeared in a paper by Naur [29] and has also been studied widely [10,12,13]. 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 [12], containing five errors. Three of these (1, 3, and 4 in their numbering) are caught by mutation analysis.