

## HEURISTICS FOR DETERMINING EQUIVALENCE OF PROGRAM MUTATIONS

Douglas Baldwin

and

Frederick Sayward

Department of Computer Science  
Yale University  
New Haven, Connecticut 06520

ABSTRACT

A mutant of a program  $P$  is a program  $M$  which is derived from  $P$  by making some well-defined simple change in  $P$ . Some initial investigations in the area of automatically detecting equivalent mutants of a program are presented. The idea is based on the observation that compiler optimization can be considered a process of altering a program to an equivalent but more efficient mutant of the program. Thus, the inverse of compiler optimization techniques can be seen as, in essence, equivalent mutation detectors.

1.0 INTRODUCTION

A mutant of a program  $P$  is defined as a program  $P'$  derived from  $P$  by making one of a set of carefully defined syntactic changes in  $P$ . Typical changes include replacing one arithmetic operator by another, one statement by another, and so forth. Program mutation has been used by DeMillo, Lipton and Sayward as the basis for an interactive program testing system [2]. The theory behind this system is that a set of test data  $T$  adequately tests a program  $P$  if all mutants of  $P$  are distinguished from  $P$  by either failing to produce any result or producing a different result for some element of  $T$ . On the other hand, if a mutant performs identically to  $P$  then either  $T$  does not fully test the program and further cases must be developed, or the mutant is equivalent to  $P$ . Obviously it is impossible to develop test data that distinguish between

equivalent forms of the same program, and thus it is desirable that equivalent mutants be excluded from the testing process. Unfortunately, user recognition of equivalent mutants has proven to be a difficult and tedious task. Thus it is important that the system aid the user by either automatically detecting equivalent mutants or by posing questions which provide insights on how to do so.

Our goal is to develop heuristics by which equivalent mutants can be recognized. The heuristics are primarily derived from techniques used to optimize compiler code, since the process of optimizing compiler code can be thought of as producing a series of mutants which are equivalent to the original program. It is thus expected that some of the tests developed to determine when an optimization is equivalence preserving can be applied to determine when a mutation is equivalence preserving.

Once a body of heuristics has been developed to detect equivalence of mutants it will be possible to develop a program to actually recognize them in a program testing system. This system will probably be very similar to the optimization phase of a compiler. It will generate some representation of each mutant which can be easily manipulated and apply the heuristics described below to determine if it is equivalent to the original. If so then the mutant will be flagged as equivalent and will be excluded from future testing runs.

## 2.0 PROGRAM MUTATION

As defined above a mutant of a program is a second program derived from the first through carefully defined syntactic transformations. Program mutation is the process of forming mutants from an input program.

The work described here is intended to find ways of determining equivalence of mutants derived as part of a process for testing FORTRAN programs on the EXPER [4] testing system. The mutations made by EXPER are chosen so as to duplicate as closely as possible the mistakes which a good programmer might make in coding a FORTRAN program. Thus many of the mutants involved, such as DO-loop end replacement, are specific to FORTRAN. The mutations of interest are described below:

1. Constant Replacement: Replacement of a constant, C, with C+1 or C-1.  
Ex: A=1 becomes A=0.
2. Scalar Replacement: Replacement of one scalar by another.  
Ex: A=B becomes A=C.
3. Scalar for Constant Replacement: Replacement of a constant with some scalar variable  
Ex: A=2 becomes A=B.
4. Constant for Scalar Replacement: Replacement of some scalar variable with a constant.  
Ex: A=B becomes A=2.
5. Source Constant Replacement: Replacement of one constant in the program with some other constant found in the program.  
Ex: A=3 becomes A=1 where the constant 1 appears in some other statement.
6. Array Reference for Constant Replacement: Replacement of a constant with an array reference.  
Ex: A=1 becomes A=B(1).
7. Array Reference for Scalar Replacement: Replacement of a scalar reference with an array reference.  
Ex: A=B becomes A=C(1).
8. Comparable Array Name Replacement: Replacement of a reference to one array with a reference to the same element of another array of the same size and shape.  
Ex: A=B(1,3) becomes A=X(1,3).
9. Constant for Array Reference Replacement: Replacement of an array reference with a constant.  
Ex: A=B(1) becomes A=3.
10. Scalar for Array Reference Replacement: Replacement of an array reference with a reference to a scalar.  
Ex: A=B(1) becomes A=C.

11. Array Reference for Array Reference Replacement: Replacement of one array reference by another.  
Ex: A=B(1) becomes A=C(2).
12. Unary Operator Insertion: Insertion of one of the unary operators ! (absolute value), - (negation), ++ (increment by 1) or -- (decrement by 1) in front of any data reference.  
Ex: A=B becomes A=-B.
13. Arithmetic Operator Replacement: Replacement of one arithmetic operator (+, -, \*, /, \*\*) with another.  
Ex: A=B+C becomes A=B-C.
14. Relational Operator Replacement: Replacement of one relational operator (.EQ., .LE., .GE., .LT., .GT., .NE.) with another.  
Ex: IF(A.EQ.B) GOTO 1 becomes IF(A.NE.B) GOTO 1.
15. Logical Connector Replacement: Replacement of one logical connector (.AND., .OR.) with the other.  
Ex: A.AND.B becomes A.OR.B.
16. Unary Operator Removal: Deletion of any unary operator.  
Ex: A=!B becomes A=B.
17. Statement Analysis: Replacement of any statement with a trap statement whose execution causes immediate failure of the program.  
Ex: GOTO 2 becomes CALL TRAP.
18. Statement Deletion: Removal of any statement.  
Ex: GOTO 2 is removed, i.e. becomes CONTINUE.
19. Return Statement Replacement: Replacement of any statement by a RETURN statement.  
Ex: A=0 becomes RETURN.
20. Goto Statement Replacement: Replacement of any GOTO statement with a GOTO to a different label.  
Ex: GOTO 1 becomes GOTO 3.
21. DO Statement End Replacement: Replacement of the end label in a DO statement with some other label.  
Ex: DO 2 I=1,10 becomes DO 1 I=1,10.
22. Data Statement Alteration: Changing the values assigned by a DATA statement.  
Ex: DATA A /2/ becomes DATA A /1/.
23. Unary Operator Replacement: Replacement of one unary operator by another.  
Ex: A=!B becomes A=++B.

Obviously some of the mutations described above can produce mutants which are equivalent to the original program. For instance, replacing  $A=0$  with  $A=10$  does not change a program. It might be hoped that detection of equivalent mutants would be easy, since the mutations involved are so simple and well defined. Unfortunately this is not the case. It is easily shown that the general problem of determining the equivalence of two primitive recursive functions is undecidable [1]. If we let P1 and P2 be FORTRAN routines corresponding to two arbitrary primitive recursive functions we can show that the equivalence of mutants is undecidable. Consider the following program to which the mutation "GOTO Statement Replacement" has been applied:

```

      GOTO 1
1     P1
      STOP
2     P2
      STOP

```

The resulting mutant looks like:

```

      GOTO 2
1     P1
      STOP
2     P2
      STOP

```

Plainly these programs are equivalent if and only if P1 and P2 are equivalent. Since the equivalence of P1 and P2 is undecidable, the equivalence of the mutant and original programs must also be undecidable.

The easiest way to show that two programs are not equivalent is to find some input on which they produce different outputs. This is the basic function of EXPER as a program testing tool, and thus many mutants do not need to be tested for equivalence. At any given stage those mutants which produce the same output as the original program on all test

data are called live mutants. Obviously it is only the live mutants to which sophisticated equivalence tests must be applied at all. Since the equivalence problem for program mutants is undecidable, any equivalence testing process will not always be able to detect all equivalent mutants. Thus the final decision about whether a mutant is equivalent to the original program might have to be left to the user. The goal of the testing process should be to make one of three decisions about any mutant:

1. It is definitely equivalent to the original program.
2. It might be equivalent to the original program, but the information needed to make this determination is not completely available. The system should identify the needed information and ask the user to supply it.
3. None of the known tests are able to determine whether the mutant and the original are equivalent. The system is unable to help the user at all.

### 3.0 OPTIMIZATION TECHNIQUES

Almost all of the techniques used in optimizing compiler code can be applied in some way to decide whether a mutant is equivalent to the original program. Some are useful only in very limited sets of situations, whereas others can be applied to many types of mutation. All the techniques discussed below can be applied widely enough that it would be worthwhile to implement them in an actual equivalence tester.

The easiest way to implement these techniques is in conjunction with a flow graph of the program being mutated. A flow graph is a directed graph in which each node represents a statement or group of statements through which program control flows linearly (basic blocks). Thus any

node in the flow graph represents a fragment of code which is entered only at the first statement of the block and exited only from the last. Furthermore there are no loops or branches within the node. The edges of the flow graph represent branches within the program from one basic block to another. Efficient algorithms exist for generating flow graphs from programs, for instance the process outlined by Schaefer ([5], pages 12-20). Thus it is reasonable to expect such a representation to be available to the equivalence tester. Furthermore, since mutants are so similar to the program from which they are derived, it will be easy to derive the flow graph of the mutant directly from the flow graph of the original in most cases. In the discussion below it is assumed that the equivalence tester can examine programs at the statement and token level; whether these entities are individual nodes in the flow graph or packed many per node is irrelevant.

The various optimization techniques which seem applicable to testing mutant equivalence are listed below.

### 3.1 Constant Propagation

Constant propagation involves replacing expressions involving constants with other constants to eliminate run-time evaluation. Generally the compiler keeps track as far as possible of the value of each variable throughout the program. At any point where an expression involves only variables whose values are known the result of the expression can be computed at compile time and placed in the program as a new constant. Thus this optimization applied to the code fragment

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

would produce the equivalent code

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

An elegant scheme for global program analysis is given by Kildall [3]. This scheme associates with each statement of the program a pool of data which are being propagated through the program. Such data pools can be used for constant propagation by letting the elements of the pool be ordered pairs whose first element represents a variable and whose second element represents a value. Other applications of this approach to program analysis are discussed below. This scheme is ideally suited to the needs of an equivalence tester.

### 3.2 Invariant Propagation

Invariant propagation is similar to constant propagation in that it involves associating with each statement of the program a set of invariant relationships between data elements. For instance, invariant propagation will note such things about a program as "X<0" or "B=1". As indicated by the last example constant propagation is a special case of invariant propagation. This technique is of limited use in compilers, but is very powerful for detecting equivalent mutants.

Invariant propagation can be implemented using Kildall's scheme for constant propagation by replacing the variable and value pairs with triples of the form <object>, <relation>, <object>. Each <object> represents either a variable or constant, and <relation> is one of the algebraic relations <, >, =, ≤, ≥, or <>. The only difficulty is that an invariant propagation algorithm should be able to replace a strong

relationship with a weaker one (i.e. replace " $A=1$ " with " $A \geq 1$ "). The propagation algorithm should also be able to apply transitivity to deduce relationships such as " $A < 0$ " from the relationships " $A < B$ " and " $B < 0$ ".

### 3.3 Common Subexpression Elimination

One of the optimizations frequently performed by compilers is to recognize subexpressions which occur many times but only need to be evaluated once. For instance, in the code fragment

```
A=X+Y
B=X+Y+Z
```

The expression " $X+Y$ " is evaluated two times. The common subexpression can be eliminated by evaluating it once and assigning the result to a temporary variable  $T$ , yielding:

```
T=X+Y
A=T
B=T+Z
```

Kildall [3] demonstrates how his scheme for global analysis can be applied to common subexpression elimination. In this application the data pools are sets of expressions which are partitioned into equivalence classes such that all expressions in equivalence class  $E$  have the same value. Thus the example above might have sets as shown below, where " $|$ " divides equivalence classes: (Note the addition of a CONTINUE statement to show the set after the assignment to  $B$ .)

```
A=X+Y      { }
B=X+Y+Z    { A,X+Y }
CONTINUE   { A,X+Y | B,X+Y+Z,A+Z }
```

Note that the algorithm described by Kildall generates equivalent

expressions which are not used in the program, such as A+Z in the same partition as X+Y+Z above. This feature allows the widest possible range of equivalent expressions to be recognized.

### 3.4 Recognition of Loop Invariants

A common optimizing technique removes code from inside loops if the execution of that code does not depend on the iteration of the loop.

Thus a loop of the form

```

          DO 1 I=1, 10
            A(I)=0
1         B=0

```

would be replaced by

```

          DO 1 I=1, 10
1         A(I)=0
          B=0

```

Since many of EXPER's mutations change the boundaries of loops, techniques for recognizing when code can be removed from a loop can be useful in detecting equivalences. Conditions for detecting operations which can be removed from loops are given by Schaefer ([5], pages 122-134).

### 3.5 Hoisting and Sinking

Hoisting and sinking are related to removal of code from loops in that they involve moving code which would be repeated several times to a place where it will only be executed once. Thus the code fragment

```

          IF(A.EQ.0) GOTO 1
          C=0
          B=2
          GOTO 2
1         C=1
          B=2
2         etc.

```

could be replaced by

```

        B=2
        IF(A.EQ.0) GOTO 1
        C=0
        GOTO 2
1       C=1
2       etc.

```

Here the assignment  $B=2$  has been hoisted to a position before the conditionally executed part of the program. Similarly sinking involves moving code to a position after some set of blocks. Mathematical rules for detecting the feasibility of hoisting or sinking are given on pages 115-119 of Schaefer [5].

### 3.6 Dead Code Detection

Dead code detection involves the identification of sections of a program which will either never be executed or whose execution is irrelevant. An example of typical dead code is the fragment below, in which the second assignment to  $A$  kills the first:

```

A=B+C
A=0

```

Schaefer [5] discusses rules for detecting dead code of this form on pages 156-161.

Another example of dead code is the case in which one or more basic blocks of a program are not connected to the rest of the flow graph. Then, as long as there is only one entrance to the program some section is never executed and can be removed entirely. This case is not expected to arise very often in programs written by humans, but mutations may easily make a large part of a program inaccessible from the entry node. For example, consider the following mutant of a program:

```

A=1
RETURN
B=A+2
  etc

```

Here the insertion of the RETURN statement has made everything between it and the next label which is referenced in a GOTO inaccessible. This type of dead code is easily detected by examining the flow graph of the program in question.

#### 4.0 APPLICATIONS

Each of the above optimization techniques can be applied to detect equivalent mutants arising from one or more of the mutations applied by EXPER. Each is discussed below.

##### 4.1 Constant Propagation

Constant propagation is most useful for detecting cases in which a mutant is not equivalent to the original program. Any mutant which could affect the known value of a variable can be detected in this fashion. The mutants most easily checked using this scheme are those involving replacement of one data reference with another (Constant Replacement, Scalar Replacement, Scalar for Constant Replacement, Constant for Scalar Replacement, Source Constant Replacement, Array Reference for Constant Replacement, Array Reference for Scalar Replacement, Array Name Replacement, Constant for Array Reference Replacement, Scalar for Array Reference Replacement, Array Reference for Array Reference Replacement, and Data Statement Alteration). Equivalences which may be detected, but with lower probability, are those involving changes to expressions (Arithmetic Operator Replacement, Unary Operator Removal, Unary Operator

Insertion, and Unary Operator Replacement). It is possible that equivalences involving actual changes to the program flow could be detected, but it should be much easier to detect these by comparing the flow graphs.

The mechanism for testing equivalence of mutants using constant propagation is as follows: At all points subsequent to the mutation compare the constant pools of the original program and the mutant. If they differ it is likely (though not certain) that the mutant is not equivalent to the original program. The following example demonstrates this form of detection:

Original Program		Mutant Program	
Code	Constants	Code	Constants
A=1		A=2	
B=A+2	(A,1)	B=A+2	(A,2)
etc	(A,1),(B,3)	etc	(A,2),(B,4)

Here a mutation has replaced the assignment of 1 to A with an assignment of 2. The change in the program is reflected in the changed constant pools following the mutation. Unless the assignments to A and B are dead it is reasonable to assume that the mutation is not equivalent to the original, and to try to develop test data which substantiate this assumption.

A firm test of non-equivalence can be made if one of the output variables appears in the constant pool for a RETURN statement. Then if the known value of this variable differs between the mutant and original programs we know that they are not equivalent, since they return different values on identical inputs. Obviously this test is valid only if some path exists from the entry node of the program being tested to the exit in question. This question can be resolved through dead code

detection.

#### 4.2 Invariant Propagation

As shown above invariant propagation is really a super-set of constant propagation, and thus it can be used to test all the sorts of mutants discussed under constant propagation. However since a great deal more information is carried by invariant relationships than by equality to a constant, this technique is far more powerful than constant propagation. It is particularly useful for testing the equivalence of mutants involving unary operators (i.e. Unary Operator Removal, Unary Operator Insertion, and Unary Operator Replacement). In many cases these operators only affect an expression if it has a certain relationship to 0. For example, taking the absolute value of an expression only changes the program if that expression evaluates to a value less than zero; negating an expression does not change anything if that expression always evaluates to 0, and so forth. These facts can be used as shown in the following example:

Original Program		Mutant Program	
Code	Invariants	Code	Invariants
IF(A.LT.0) GOTO 1		IF(A.LT.0) GOTO 1	
B=A	<u>A&gt;0</u>	B= A	<u>A&gt;0</u>

In this case the conditional allows us to determine an invariant (A>0), which in turn allows us to determine that the mutant program is equivalent to the original, since taking the absolute value of a positive quantity is a no-op.

The power of invariant propagation is vastly increased if the propagation and testing algorithms can take advantage of transitivity and replacement of one condition by a weaker one. Both of these features are

demonstrated below:

Original Program	
Code	Invariants
A=0	
CONTINUE	A=0
1 B=A	$A \geq 0, A < 5$
C=B	$A \geq 0, A < 5, B=A$
A=A+1	$A \geq 0, A < 5, B=A$
IF(A.LT.5)	$A \geq 0, A < 5, B=A$
GOTO 1	$A \geq 0, A < 5, B=A$

Mutant Program	
Code	Invariants
A=0	
CONTINUE	A=0
1 B=A	$A \geq 0, A < 5$
C=B	$A \geq 0, A < 5, B=A, C=B$
A=A+1	$A \geq 0, A < 5, B=A, C=B$
IF(A.LT.5)	$A \geq 0, A < 5, B=A, C=B$
GOTO 1	$A \geq 0, A < 5, B=A, C=B$

Note that the algorithm for generating invariant pools recognizes the loop in this program and is thus able to determine an upper bound on A. Obviously the invariants shown assume that no other branches to label 1 exist. The relation  $A=0$  is replaced with the weaker  $A \geq 0$  when the statement  $A=A+1$  is detected at the end of the loop. Applying transitivity to the mutated pair  $C=B$  and  $C=B$  allows us to decide that the mutant is equivalent to the original since  $B=A$  and  $A \geq 0$ .

There is one important feature of EXPER which is useful in generating invariant pools: EXPER can perform run-time checks of array bounds. Thus the following statements generate the invariant pool shown:

Code	Invariants
DIMENSION A(5)	
A(J)=0	$J \geq 1, J \leq 5$

Because EXPER checks array bounds any program aborts if J is less than 1 or greater than 5 in the assignment to A(J). Thus any program or mutant for which the given invariants did not hold prior to executing the

assignment would have failed, and thus would obviously not be a correct program.

#### 4.3 Common Subexpression Elimination

Kildall's equivalence partitions[3] provide an excellent way to handle mutations in assignment statements. Changing an arithmetic operator changes the expression placed in the equivalence class of the variable to which the assignment was made. Similarly, mutations which change an operand or destination in an assignment will produce changes in the equivalence classes following the assignment. Thus comparing equivalence classes can show that the mutant and original differ. As an example, consider the program and mutant shown below:

Original Program	
Code	Equivalence Classes
A=B+C	
etc.	{A,B+C}

  

Mutant Program	
Code	Equivalence Classes
A=B-C	
etc.	{A,B-C}

Comparing the two sets of equivalence classes shows that A has a different value in the two programs. As with constant propagation, we can assume that the mutant is not equivalent to the original program, and that test data should be developed to verify this assumption.

Common subexpression detection can also be used to show that a mutant is equivalent to the original program. If the mutation has changed part of an expression E to E', but E and E' are in the same equivalence class, then the mutant is equivalent to the original program. The example below demonstrates this situation:

Original Program	
Code	Equivalence Classes
A=B+C	
D=B+C	{A, B+C}
X(A+E)=0	{A, B+C, D}

  

Mutant Program	
Code	Equivalence Classes
A=B+C	
D=B+C	{A, B+C}
X(D+E)=0	{A, B+C, D}

Since A and D are in the same equivalence class we can conclude that the mutation (replacing A with D in the subscript) did not change the program. Note that since the equality of A and D is determined through assignment of a common expression this equivalence would be hard to detect using a simpler heuristic such as invariant propagation.

#### 4.4 Recognition of Loop Invariants

Many mutations change the size of loops. The most obvious of these is the DO-loop End Replacement operator, although the GOTO Replacement operator can also alter loops. In cases where a loop has been changed to include more or less code than in the original, recognition of loop invariants can be used to decide whether or not the change is significant. Examination of the flow graphs should make cases in which loops have changed fairly easy to detect; thus it is easy to decide when to apply these tests. The basic application simply involves deciding whether or not the excess code (that is, the code which does not appear in both loops) is loop invariant. If it is then the expansion (or contraction) of the loop has not changed the outputs of the program. As an example, consider the following code:

Original Program	Mutant Program
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

The mutation above expanded the DO-loop to include the assignment of 0 to B. Since this assignment is loop-invariant it does not matter whether it is done 10 times inside the loop or 1 time outside it. Thus the original and mutant programs are equivalent.

#### 4.5 Hoisting and Sinking

These tests are used in situations similar to those in which testing of loop-invariants is used, except that they apply to cases in which the code skipped or included by a branch is changed. Candidates for this sort of change include GOTO Replacement and Statement Deletion. In these cases the mutant and original programs are equivalent if the code added to or removed from a basic block can be hoisted or sunk out of that block. Consider the following example:

Original Program	Mutant Program
IF(A.EQ.0) GOTO 1	IF(A.EQ.0) GOTO 2
A=A+1	A=A+1
2 B=0	2 B=0
GOTO 3	GOTO 3
1 B=0	1 B=0
3 etc	3 etc

In this case B is set to zero regardless of whether we do it at line 2 or line 1. A more compact form is produced by hoisting the assignment to B, namely

```

B=0
IF(A.EQ.0) GOTO 3
A=A+1
3  etc

```

Because this hoisting is possible the mutant is equivalent to the

original program.

Because the code skipped by the statement "GOTO 3" can be hoisted the branch is unnecessary. Thus the hoisting test will also show that the mutant derived by deleting this branch is equivalent to the original program.

#### 4.6 Dead Code Detection

As mentioned above this test is very important in guaranteeing the reliability of tests based on invariant propagation (including constant propagation). It can also be used to test the equivalence of some mutants in its own right. The equivalences which are most likely to be detected by this technique are those arising from mutations that alter the flow graph in some way. Such mutants include Statement Analysis (since this mutant replaces any statement with an abnormal exit), Statement Deletion (if GOTO or RETURN statements are deleted), Return Statement Replacement, and GOTO Replacement.

The best way to use dead code detection to test mutants of this form is to examine the flow graphs of the two programs. If any node appears in the mutant which is not connected to the rest of the graph it is reasonable to expect that the mutant is not equivalent to the original. (The only exception being the case in which the disconnected node consists only of dead assignments. This situation is discussed in general below). An example involving Return Statement Replacement is shown below:

Original Program		Mutant Program	
Code	Flow Graph	Code	Flow Graph
A=1		A=1	
B=2		RETURN	
C=3		C=3	

The RETURN statement has broken the original single node into 2 nodes with no connection between them. Thus one can conclude that since code which is executed in the original program (assuming the node is accessible in the first place) is not executed in the mutant, the two are different.

A slightly different application of dead code detection involves making sure that mutated code is not inaccessible or dead in the first place. If it is then the mutant must be equivalent to the original program. This application is identical to the application in compiler optimization where code is identified as dead and excluded from the final output. It applies to all mutant operators. An example of this sort of analysis in testing equivalence is shown below:

Original Program	Mutant Program
A=1	A=2
A=B+C	A=B+C

Here the first assignment to A is killed by the second assignment, and thus any change to its right-hand side is insignificant. A more drastic example shows inaccessible code. Again, the mutant to code which can never be executed is unimportant.

Original Program	Mutant Program
GOTO 1	GOTO 1
A=2	A=-2
1 etc	1 etc

Some cases in which a mutation has killed a block of code can be detected by using invariant propagation. The program fragment shown below shows how this can happen:

Original Program		
Code		Invariants
IF(A.GT.B) GOTO 1		
FLAG1=.TRUE.		$A < B$
IF(A.LT.B) GOTO 2		$A < B$
FLAG2=.TRUE.		$A = B$
2     etc		$A < B$

  

Mutant Program		
Code		Invariants
IF(A.GT.B) GOTO 1		
FLAG1=.TRUE.		$A < B$
IF(A.LE.B) GOTO 2		$A < B$
FLAG2=.TRUE.		
2     etc		$A < B$

Here the mutation has replaced the test  $A < B$  with the test  $A \leq B$ . However, the invariant pool tells us that A is always less than or equal to B, and thus the branch will always be taken, and the assignment to FLAG2 is dead. Note that without knowing the relationship between A and B it is impossible to determine that this assignment is dead.

#### 5.0 AN EQUIVALENCE TESTING POST-PROCESSOR FOR EXPER

The above ideas for determining equivalence can be applied in a post-processor to EXPER in order to reduce the time spent by the user dealing with equivalent mutants. This processor should be run after the mutants have been executed on the test data, since experience shows that as many as 90 per cent of the mutants can be eliminated on the first testing run. Of the remaining mutants, those which are found by the post-processor to be equivalent are flagged as such and the user need not consider them further. Only those which are not found to be equivalent

are analyzed by the user to improve his test data. At any point the user can manually over-ride the post-processor by declaring a live mutant to be equivalent to the original program or by declaring one that was thought to be equivalent to be live again.

The analysis proceeds much as it would in a compiler, with a few exceptions which arise due to the fact that we do not necessarily want to produce efficiently optimized code. For instance, it is not important that we worry about compiler-generated constants, since they can never be mutated.

The first step is to express the original program as a flow graph, as discussed above. This step may be done as part of EXPER's parsing or other processing of the program. As each live mutant is tested for equivalence to the original program a flow graph is generated for it. In many cases this flow graph will be isomorphic to the original so that only the contents of one node need to be modified. In more complex cases, where the shape of the flow graph is changed, the mutant's flow graph can still be derived from the original. EXPER represents mutants as a descriptor record describing the change made to the original program. These records fully describe the mutant, and thus allow the mutant's flow graph to be derived without re-generating it from a source program.

Just as it is expected that mutant flow graphs can be efficiently derived from the original flow graph, it is also expected that the invariant and common expression pools described above will not have to be computed for each mutant. Instead, the pools for the original can be computed at parse time and the mutant's pools derived from them. As

suggested above, many mutations cause a relation to change, move an expression from one equivalence class to another, or make similarly limited changes in the pools. These changes can be easily detected using the descriptor record of the mutant, and can be made as local modifications to the pools. Obviously, care will have to be taken that any side effects of these local changes are detected, but doing so should be significantly less expensive than regenerating the entire pool.

The invariant and common expression pools described above can be combined into a single pool by replacing the individual variables or constants involved in invariant relationships with the equivalence class sets used to recognize common expressions. Note that using this scheme the relationships "equal to" and "not equal to" do not need to be explicitly represented, since if two objects are in the same set they must be equal, whereas if they are not in the same set they must be unequal. If the entire structure of sets and relationships is represented as a directed graph whose nodes correspond to sets and whose edges to relationships (obviously the edges must be labelled as to what relationship) then the problem of applying transitivity becomes one of simply following either edges labelled '>' and '>=' or edges labelled '<' and '<=' until either the desired relationship is derived or no edges with the appropriate labels remain. Note that no cycles can occur which involve such paths. Assume such a cycle did exist, for instance a path using only edges marked '<' or '<=' from node A to node B and back to node A. Since a path from A to B exists, transitivity implies that for any X in A and Y in B,  $X < Y$ . However, because a path from B to A exists we also have the statement  $Y < X$ . Because X and Y are in different sets we know that X is not equal to Y, and thus the derived relationships are

contradictory.

Representing the pools in this manner allows a great deal of flexibility in testing equivalences. The following example shows how this can happen:

Original Program	
Code	Invariant & Expression Pool
A=B+C	
D=E+F	{A,B+C}
IF(B+C.LE.D) GOTO 1	{A,B+C},{D,E+F}
X(A+G)=0	{A,B+C}>{D,E+F}
etc.	

Mutant Program	
Code	Invariant & Expression Pool
A=B+C	
D=E+F	{A,B+C}
IF(B+C.LE.D) GOTO 1	{A,B+C},{D,E+F}
X(D+G)=0	{A,B+C}<{D,E+F}
etc.	

In this example the conditional branch allows a relationship between B+C and D to be deduced. Because the relationship is then applied to all elements equal to either B+C or D we can conclude that replacing A with D in the subscript yields a mutant subscript which is always greater than the original subscript. This fact suggests that the mutant is not equivalent to the original.

Once the modified invariant pool described above is formed it is used to aid the detection and removal of dead code. Once dead code has been removed the mutant and original are compared to see if they are obviously equivalent. If so, the mutant is placed in the equivalent mutants pool and not processed further.

Since dead code is irrelevant to the state of the program, removing it will not make the invariant pools incorrect. However, it may be possible that removing dead code enables invariant conditions to be strengthened. The following example shows how this can happen:

Original Program	Mutant Program
A=0	A=0
IF(C.GT.D) GOTO 2	IF(C.GT.D) GOTO 2
IF(C.LT.D) GOTO 1	IF(C.LE.D) GOTO 1
A=A+1	A=A+1
1        etc	1        etc.

The mutation above is a case in which changing a conditional (C.LT.D became C.LE.D) kills a block of code. The section of code killed is the increment of A. Because of this increment the strongest statement that can be made about A at label 1 is  $A > 0$ . Because the increment of A is dead in the mutant this invariant can be tightened to  $A = 0$ , assuming no other branches to label 1 exist.

Those mutants which have not been eliminated by manipulation of the flow graphs are then tested for equivalence based on loop invariants or the possibility of hoisting. Any equivalences thus found are placed in the equivalent mutants pool. Again, it is often possible to apply these tests to the original program at parse time and deduce their results on a mutant from the mutant's descriptor record. Only rarely will it be necessary to actually test the mutant.

The final phase of the post-processor applies the invariant pools generated in the first phase to actual detection of equivalent mutants. In this phase many mutants may be automatically eliminated, especially those involving unary operators. This is also a convenient place to provide user interaction in the equivalence determining process. The

processor would be driven by a set of rules describing sufficient conditions for equivalence of a mutant to the original. For instance, there might be a rule concerning absolute values which can be conceptualized as "Insertion of absolute value preserves equivalence if its argument is greater than or equal to 0". When the processor is unable to decide whether a rule is applicable by itself, it turns to the user for help. This help is requested by forming a question from the rule and posing this question to the user. For example, if an absolute value operation has been inserted in front of a variable which does not appear in the invariant pool for that statement the processor could prompt "Is X always greater than or equal to 0?". If the user replies in the affirmative the mutant is flagged as equivalent.

#### 6.0 REMARKS

It has been shown above how many techniques from compiler optimization can be applied to detect equivalent mutants of a program. Several areas remain to be explored however.

In the EXPER system only first order mutations are considered (i.e. mutants coming from one program change), but conceivably some higher order mutants may be worthy of consideration. In many cases the heuristics described here can be extended very easily to detect equivalent mutants of higher order. It is also true that in many cases equivalence can be tested transitively, i.e. if program P is equivalent to P' and P' is equivalent to P'' then P is equivalent to P''. However, it is often true that a high-order mutant can be equivalent to some program without having intermediate mutants equivalent to either. For

instance the following program fragments are equivalent:

```
IF(I.EQ.1) GOTO 1
```

and

```
IF(--I.EQ.0) GOTO 1
```

However, neither is necessarily equivalent to either of the intermediate mutants

```
IF(I.EQ.0) GOTO 1
```

or

```
IF(--I.EQ.1) GOTO 1
```

Fortunately the problem of equivalence of high order mutants is not a serious problem because of the Coupling Effect: Test data that screens out all first order mutants will screen out all higher order mutants [2]. Thus only first order mutants need to be considered in evaluating test data

A more interesting problem involves the detection of equivalences which are very dependent on the form in which the programmer has chosen to express his algorithm. As an example consider the fragment below which tests whether or not a number N is prime.

```
IF(N.LE.2) GOTO 3
L=N-1
DO 1 I=2,L
IF(N.EQ.(N/I)*I) GOTO 2
1 CONTINUE
3 PRIME=.TRUE.
RETURN
2 PRIME=.FALSE.
RETURN
```

It is really only necessary to let the DO loop run from 2 to  $\text{INT}(\text{SQRT}(N))$ . The test  $N.LE.2$  means that only N greater than or equal to 3 will be used as upper limits for the loop. Since  $\text{INT}(\text{SQRT}(3))-1$ ,  $\text{INT}(\text{SQRT}(N)) \leq N-2$ . Thus the mutation which replaces L with --L in this

loop is equivalent to the original. Because the equivalence of this mutant is so closely related to the conceptual nature of the program it seems very difficult to automatically prove it. This problem might be solved through the interactive part of the post-processor. Specifically, it is easy to find out where the mutant occurred, and the processor could simply ask "Is it acceptable for this loop be executed from 2 to L-1?".

Several techniques for detecting equivalent mutants have been described. These techniques should be capable of finding a significant number of cases in which a mutant is equivalent to the original program, since experience indicates that most equivalences are very simple ones. Often they involve the insertion of the absolute value operator, a case that is particularly easy to detect using invariant propagation. More complex equivalences can be tested interactively with the user. The questions thus posed should help the user decide whether or not to manually declare a mutant equivalent to the original program.

Several questions concerning equivalence detection remain open. At several points in the above discussion it is asserted that the data needed to determine equivalence (e.g. flow graphs, invariant pools, etc.) can be derived efficiently from the corresponding data for the original program and the mutant's descriptor record. While these assertions are undoubtedly true in many cases, exactly how often remains unknown. Further experimentation is required in this area, particularly with regard for the following questions:

1. In what fraction of the cases is it necessary to generate a flow graph for a mutant from scratch?
2. In what fraction of the cases is it necessary to regenerate the invariant pools for a mutant?

3. It is unlikely that a change to an invariant pool will affect only that pool. On the average, how many pools will be affected? How does the cost of determining all affects compare to the cost of re-computing the invariant pools?

#### REFERENCES

1. Davis, Martin Computability and Unsolvability (McGraw-Hill Co., New York, New York: 1958).
2. DeMillo, Richard A.; Lipton, Richard J.; and Sayward, Frederick G. "Hints on Test Data Selection: Help for the Practicing Programmer" reprinted from Computer 11, 4 (April 1978), pp. 34-43.
3. Kildall, Gary A. "A Unified Approach to Global Program Optimization" in Conference Record of ACM Symposium on Programming Languages, pp. 194-205, 1973.
4. Lipton, Richard J. and Sayward, Frederick G. "The Status of Research on Program Mutation", reprinted from Digest for the Workshop on Software Testing and Test Documentation, Dec. 1978, pp. 355-373.
5. Schaefer, Marvin. A Mathematical Theory of Global Program Optimization (Prentice Hall, Englewood Cliffs, N.J., 1973)