Using Compiler Optimization Techniques to Detect Equivalent Mutants

A. Jefferson Offutt *
ISSE Department
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
email: ofut@isse.gmu.edu

W. Michael Craft
AT&T Global Information Solutions
3245 Platt Springs Rd
West Columbia
SC 29170

December 27, 1996

Abstract

Mutation analysis is a software testing technique that requires the tester to generate test data that will find specific, well-defined errors. Mutation testing executes many slightly differing versions, called *mutants*, of the same program to evaluate the quality of the data used to test the program. Although these mutants are generated and executed efficiently by automated methods, many of the mutants are functionally *equivalent* to the original program and are not useful for testing. Recognizing and eliminating equivalent mutants is currently done by hand, a time-consuming and arduous task. This problem is currently a major obstacle to the practical application of mutation testing.

This paper presents extensions to previous work in detecting equivalent mutants; specifically, algorithms for determining several classes of equivalent mutants are presented, discuss an implementation of these algorithms, and present results from using this implementation. These algorithms are based on data flow analysis and six compiler optimization techniques. Each of these techniques is described and how they are used to detect equivalent mutants. The design of the tool and some experimental results using it are also presented.

Keywords—compiler optimizations, software testing, mutation testing, experimental software engineering.
The Journal of Software Testing, Verification, and Reliability, Wiley, vol 4, no 3, September 1994, pp. 131–154.

1 INTRODUCTION

A testing *criterion* selects a finite set of test cases that, if executed successfully, will provide the tester with a high level of confidence in the software being tested. Most testing criteria divide the program's input space into subsets such that every test case in the same subset has similar properties. Then the program can be tested using one test case from each subset. For example, statement coverage divides program inputs into subsets where each test case in a subset will cause the same statement to be reached.

Fault-based testing is a general strategy for developing test data that divides test data into subsets that will detect the same general kinds of faults. *Mutation testing* [DLS78, DO91, Ham77] is one such fault-based testing method.

*Supported in part by the National Science Foundation under grant CCR-93-11967. Much of this work was done while the authors were with Clemson University.

Original Program		With Embedded Mutants	
	INTEGER FUNCTION Min (I,J)		INTEGER FUNCTION Min (I,J)
	INTEGER I, J		INTEGER I,J
1	Min = I	1	Min = I
2	IF (J .LT. I) Min = J	$\Delta 1$	Min = **J**
3	RETURN	2	IF (J .LT. I) Min = J
		$\Delta 2$	IF (J .**GT**. I) Min = J
		$\Delta 3$	IF (J .LT. I) **TRAP**
		$\Delta 4$	IF (J .LT. **Min**) Min = J
		3	RETURN

Figure 1: Function MIN

1.1 Mutation Testing Overview

Mutation testing helps users iteratively create sets of test data by interacting with the users to strengthen the quality of the test data. During mutation testing, faults are introduced into programs by creating many versions of the software, each containing one fault. Test cases are used to execute these faulty programs with the goal of causing each faulty program to produce incorrect output (*fail*). Hence the term mutation; faulty programs are *mutants* of the original, and a mutant is *killed* when it fails. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process because the faults represented by that mutant have been detected.

Figure 1 shows four mutations; the original program is shown to the left, and the mutant programs are represented on the right by the lines preceded by the Δ symbol. Note that each mutated statement represents a separate program. A mutation *operator* is a rule that is applied to a program to create mutants. The mutants created by the same operator are said to be of the same *type*. The Mothra mutation system [DGK+88] uses 22 mutation operators to test Fortran 77 programs. The 22 mutation operators supported by the Mothra system replace each operand by each other syntactically legal operand ($\Delta 1$ and $\Delta 4$ in Figure 1), modify expressions by replacing operators and inserting new operators ($\Delta 2$), and modify entire statements ($\Delta 3$). The mutation operators used by the Mothra system are shown in Table 1.

The mutation testing process begins with an automated mutation system creating the mutants of a test program. Test cases are then added, either manually or automatically, to the mutation system and the user checks the output of the program on each test case to see if it is correct. If incorrect, a fault has been found and the program must be modified and the process restarted. If the output is correct, that test case is executed against each live mutant. If the output of a mutant differs from that of the original program, it is incorrect and the mutant is killed.

After each mutant has been executed with each test case, each remaining mutant falls into one of two categories. One, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, a new test case needs to be created. Two, the mutant is functionally *equivalent* to the original program. An equivalent mutant will always produce the same output as the original program, so no test case can kill it. Once identified as equivalent, there is no need for the mutant to remain in the system for further

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	`DO` statement end replacement
DSA	`DATA` statement alterations
GLR	`GOTO` label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	`RETURN` statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Table 1: **Mothra Mutation Operators for Fortran 77.**

consideration.

1.2 Equivalent Mutants

The last mutant in Figure 1 ($\Delta 4$) is equivalent. Note that the reference to **I** on statement 2 has been replaced by a reference to **Min**. The value of **I** has been assigned to **Min** at statement 1, thus these two integer variables always have the same value at this point in the program and the replacement has no effect on the functional behavior of the program. Thus the output of the mutated program will always be identical to that of the original.

Previous mutation systems required equivalent mutants to be recognized by human examination, making it one of the most expensive parts of the mutation process. This paper describes algorithms that partially solve the problem of detecting equivalent mutants. These algorithms are based on suggestions by Baldwin and Sayward [BS79]; we have designed and implemented algorithms that extend their suggestions. In section 2, the problem is examined, and previous work done on this problem is presented. Six techniques for partially solving this problem involving data flow analysis and compiler optimization strategies are presented in section 3. Because finding complete algorithms in the literature is difficult, and because the algorithms here differ significantly from the standard compiler algorithms that they are based on, it has been chosen to offer the complete algorithms in pseudo-code form. An automatic equivalent mutant detector, the Equalizer,

is presented in section 4, and a description of several experiments using the Equalizer is given in section 5. Finally, suggestions for further research are presented in section 6 and concluding remarks are in section 7.

2 DETECTING EQUIVALENT MUTANTS

Convincing oneself that a mutant is equivalent is a complicated and arduous task that requires an in-depth analysis and understanding of the program. Budd and Angluin [BA82] examine the relationships between equivalence and test data generation. They show that if there is a computable procedure for generating mutation-adequate test data for a program, there is also a computable procedure for checking if the program is equivalent to the mutants and vice versa. They also show that, for many sets of mutation operators, neither of these problems is decidable. Thus, there can be no complete algorithmic solution to the mutation equivalence problem.

Fortunately, the mutation equivalence program has two advantages over the general equivalence problem. First, equivalence of arbitrary pairs of programs does not have to be determined. Because of the definitions of the mutation operators, mutant programs are very much like their original program (Budd and Angluin describe mutants as "neighbors" of the original program). We can take advantage of this fact to develop techniques and heuristics for detecting many of the equivalent mutants. Second, software testing is inherently an imperfect science, thus partial results can be very valuable to the practical tester, and are often sufficient.

2.1 Distribution of Equivalent Mutants by Operator Type

It has been noted that equivalent mutants are not evenly distributed among the 22 mutation types. In fact, the equivalent mutants tend to cluster among only a few types. Table 2 summarizes statistics from the programs used in section 5 of this paper. The first column in the table gives mutation operator names and the second column gives the percentage of equivalent mutants that are created by that operator. The third column gives the percentage of all mutants that are equivalent to that type. It is interesting to note that one mutant type, *absolute value insertion* (`ABS`), accounts for over half of all equivalent mutants. The `ABS` mutation operator inserts three unary operators before each expression; ABS computes the absolute value of the expression, NEGABS computes the negative of the absolute value, and ZPUSH kills the mutant if the expression is zero, otherwise the value of the expression is unchanged.

2.2 Detecting Equivalent Mutants by Hand

It is obvious that detecting equivalent mutants automatically can save much time and energy for the testers, but Acree [Acr80] found that it could also prevent people from making errors in marking equivalent mutants. In a study of 50 mutants, half of which were equivalent, Acree found that people judged mutant equivalence correctly only about 80% of the time. The people marked equivalent mutants non-equivalent (type 2 errors) 12% of the time and non-equivalent mutants equivalent (type 1 errors) 8% of the time. Because type 2 errors can be corrected during later testing, it is really only type 1 errors that require attention. The advantage of using automated techniques to detect equivalent mutants is that the techniques can be designed so that

Mutant Type	Percent of Equivalent Mutants	Percent of All Mutants
Absolute Value Insertion	54.3	3.40
Scalar for Constant Replacement	16.1	1.70
Array for Constant Replacement	11.2	0.25
Array for Scalar Replacement	3.9	0.19
Scalar Variable Replacement	3.1	0.18
Unary Operator Insertion	3.0	0.15
Relational Operator Replacement	2.4	0.07
All Other Mutation Operators	6.0	0.30
TOTAL	100.0	6.24

Table 2: Equivalent Mutant Percentages

any mistake could be of type 2. An automated tool (if implemented correctly) should not conclude that a killable mutant is equivalent.

3 COMPILER OPTIMIZATION TECHNIQUES

Baldwin and Sayward [BS79] originally suggested the idea of using compiler optimization strategies to detect equivalent mutants. Their technical report did not specify how to do this, nor did they try to implement their suggestions. These ideas have been developed by working out how to detect equivalent mutants, and by designing and implementing algorithms. The key intuition behind this approach is that many equivalent mutants are, in some sense, either optimizations or **de**-optimizations of the original program. The transformations produced from code optimizers result in equivalent programs. When an equivalent mutant satisfies a code optimization rule, algorithms can detect that the mutant is in fact equivalent. Six types of compiler optimization techniques are used, which are described in standard compiler textbooks [FL88, ASU86]. These techniques are commonly used and are standard. While some more advanced methods may improve the application of some of these techniques, the techniques described in this paper seem to give the most benefit. The six techniques are:

1. Dead Code Detection,
2. Constant Propagation,
3. Invariant Propagation,
4. Common Subexpression Detection,
5. Loop Invariant Detection, and
6. Hoisting and Sinking.

These six techniques are described in the rest of this section, with particular emphasis on how they are adapted and applied to the current problem. Because these techniques depend on a data flow analysis of the program, some of the basic concepts of data flow analysis are first presented.

3.1 Data Flow Analysis

Data flow is a well-known program analysis technique used for compiler optimization and software testing. This paper uses the standard terminology [AC76, FL88]. A variable is *defined* (a `def`) when it is assigned a value, e.g., it appears on the left hand-side of an assignment statement. A variable is *used* when it appears in the right hand-side of an assignment (a *computation-use*) or in the expression of a branch statement (a *predicate-use*). A `def` of a variable *reaches* a `use` if there is a path in the program from the `def` to the `use` that contains no intervening definitions.

In data flow analysis, the program is first partitioned into *basic blocks*, which are maximal linear sequences of code having one entry point (the first instruction executed) and one exit (the last instruction executed). Given this partitioning of the program, the program flow of control can be represented as a directed graph in which the basic blocks are nodes and the actual flows of control are the edges.

After the basic blocks and the control flow between these blocks have been established, reaching definitions are determined by finding the set of definitions of each data item that reach each basic block. This is the union of the set of definitions that are available from those nodes that immediately precede each node. This information can be derived by using a basic reach algorithm (such as given in Allen and Cocke [AC76]) and stored in a *reach table*.

After the reach information for the blocks is determined, computing which `defs` reach a `use` is straightforward. If there exists a definition of the data item being referenced between the start of the block and the actual use, that last definition is the only reaching definition. Otherwise, each definition of the data item that reaches the beginning of the block reaches the use of that data item. With this information, exactly which definitions of a variable can be current at each use of that variable can be determined. The information gathered about each definition, in conjunction with the reach information, can now be used to determine equivalent mutants.

A common problem with data flow-based approaches is that of aliasing. A variable can be *aliased* to another variable if they both refer to the same memory location [FL88]. This can be done when two actual parameters are the same, thus aliasing the formal parameters, or when an array is accessed beyond its legal bounds, aliasing the array location with another variable. While aliasing typically makes optimization and test data generation problems more difficult, aliasing has an almost inverse effect on this work. Several of the equivalent mutant detection techniques rely on recognizing that certain variables have the same value at some point during execution. Thus, the aliasing problem can cause us to miss detection of equivalent mutants, rather than mislabeling killable mutants as equivalent. Partial solutions to the aliasing problem would improve the ability to detect equivalent mutants.

3.2 Equivalencing Mutants Using Dead Code Detection

A statement that can never be executed or whose execution is irrelevant is considered *dead*. The most obvious form of dead code is an unreachable statement, which has no control flow path from the beginning of the program to the statement. Code can be unreachable either *statically*, meaning there is no path to the statement, or *dynamically*, meaning there are no paths that can be executable. The static form for this case

is easy to detect using a control flow graph because the statement appears in a node that is unreachable from the start node. This paper does not address the dynamic form; although ongoing research is currently addressing this problem [Pan94].

Such a node can easily be detected by traversing the flow graph starting from the start node – unvisited nodes will represent dead code. Obviously any mutation that changes dead code can never affect the output of the program and is therefore equivalent.

The second form of dead code is the *dead definition*, which is a definition of a data item that is either redefined before it is referenced, or is never referenced. One restriction on this definition is that the execution of the assignment statement does not alter the value of any other data item other than the one being defined (that is, no side-effects). Any mutation of a statement that has a dead definition will be equivalent.

3.3 Equivalencing Mutants Using Constant Propagation

Constant propagation involves detecting definitions whose values are constant and can be computed at compile time. The constant propagation algorithm in this paper is modeled after the standard procedures [All69, FL88]; however, they have been extended to propagate constants not only within basic blocks but also across basic blocks. Thus, the constant definitions detected in one block are used to detect constant definitions in other blocks. This is accomplished by using the reach information derived from the data flow analysis in conjunction with a new *constant table*, which has one entry for each definition. If a definition is determined to have a constant value, the value is stored in that definition's constant table entry. This information is used to determine equivalent mutants when a mutant cannot be killed if a variable has the value in its constant table entry. The algorithm for constant propagation is given in Figure 2.

After this procedure ends, the information about each definition stored in the constant table can be used to detect equivalent mutants. The current implementation detects equivalent mutants of the types *ABS, SVR, UOI, AOR, CSR, SCR*, and *ROR*. In order to determine equivalence, the following conditions for each mutant type must hold:

- *ABS*: As described in section 2, this mutation operator has three variations, *ABS*, *NEGABS*, and *ZPUSH*. The condition for *ABS* is that the variable it acts upon is known to be constant, and its value is greater than or equal to zero. For *NEGABS*, the variable it acts upon is known to be constant, and its value is less than or equal to zero. For *ZPUSH*, the variable it acts upon is known to be constant, and its value is not equal to zero.

- *CSR*: the scalar variable replaced is known to be constant, and its value is equal to the constant replacing it.

- *SCR*: the scalar variable replacing the constant is known to be constant, and its value is equal to the constant it is replacing.

- *SVR*: both variables involved are known to be constant, and their values are equal.

- *AOR*: the operator being replaced is either + or −, the replacement operator is either + or −, and the

```
Algorithm Constant Propagation
input     Program P and control flow graph for P
output    Constant value (if known) for each definition (CT[])
declare   CT []         :  Constant Table of constant values, indexed by definitions
          cfg           :  control flow graph of P
          bb            :  basic block of P
          def           :  definition in P
          reach_def     :  reaching definition of a variable in P
          opnd          :  operand
          repeat_flag   :  boolean
          constant_flag :  boolean


 1 CT [d] := unknown, ∀ def in P
 2 FOREACH definition def in P
 3    IF (every operand in the assignment expression is a scalar constant) THEN
 4       evaluate the assignment expression to a value v
 5       CT [d] := v
 6 ENDFOR
 7 REPEAT
 8    repeat_flag := FALSE
 9    FOREACH basic block bb in cfg in breadth-first order
10       FOREACH definition def in bb
11          constant_flag := TRUE
12          FOREACH operand opnd in def
13             IF (opnd is not a scalar variable) or (opnd is not a scalar constant) THEN
14                constant_flag := FALSE
15             IF (opnd is a scalar variable) THEN
16                IF (CT [reach_def] ≠ unknown ∀ reach_def of opnd) AND
17                   (CT [reach_def_i] = CT [reach_def_j] ∀ i, j ≤ number of reaching definitions) THEN
18                      substitute the value CT [reach_def_1] in the assignment expression for opnd
19                ELSE
20                   constant_flag := FALSE
21          ENDFOR
22          IF (constant_flag = TRUE) THEN
23             evaluate assignment expression to a value v
24             IF (CT [d] ≠ v) THEN
25                CT [d] := v
26                repeat_flag := TRUE
27             ENDIF
28          ENDIF
29       ENDFOR
30    ENDFOR
31 UNTIL (repeat_flag = FALSE)
```

Figure 2: Constant Propagation Algorithm

	T	F	$<$	$=$	$>$	$\leq$	$\neq$	$\geq$
T	T	?						
F	?	F						
$<$			$<$	$\leq$	$\neq$	$\leq$	$\neq$	?
$=$			$\leq$	$=$	$\geq$	$\leq$	?	$\geq$
$>$			$\neq$	$\geq$	$>$	?	$\neq$	$\geq$
$\leq$			$\leq$	$\leq$	?	$\leq$	?	?
$\neq$			$\neq$	?	$\neq$	?	$\neq$	?
$\geq$			?	$\geq$	$\geq$	?	?	$\geq$

Table 3: Definition Status Decision Table

second operand of the operation is a scalar variable that is known to be constant and whose value is zero.

- *UOI*: the inserted operator is $-$, and the operand of the inserted operator is a scalar variable that is known to be constant and whose value is zero.
- *ROR*: the operands of the replaced relational operator are either two scalar variables known to be constant, or one constant and one scalar variable known to be constant.

3.4 Equivalencing Mutants Using Invariant Propagation

An *invariant* is a relation between two variables or a variable and a constant that is known to be true at a given point in a program. These invariants are separated into two categories. The first group of invariants pertains to the variable definitions contained in the program and is stored in the *definition invariant table*. The second group is a more general group that includes a separate invariant for each statement in the program. Relationships that are true at a particular statement in the program are stored in the *statement invariant table* at the corresponding statement number. This information is used to determine equivalent mutants when a mutant involves a variable that has the invariant marked in the definition invariant table. For example, to kill a variable replacement mutant, the new variable must have a value that differs from the old variable. If the *definition invariant decision table* indicates the two variables are equal, the mutant is equivalent.

Because of the large number of absolute value insertion mutants (*ABS*) that are equivalent, a valuable piece of information is the relationship between a variable and the constant zero (e.g., $X > 0$). Often, even if the variable's constant value cannot be determined, its relationship with zero can, so that information is stored as the *status* of the variable. In addition, if the variable is of type logical, knowing whether the variable is guaranteed to be either true or false at a particular statement is helpful. The relationship a variable has with zero at each of its uses can often be determined based upon the status of each of its reaching definitions, using the *definition status decision table* given in Table 3. The status of a variable at a given statement is computed from the *definition status decision table* by pairwise computing the status of each reaching definition.

If the statuses for each of two reaching definitions are $\mathbf{X > 0}$ and $\mathbf{X \geq 0}$, the table will return the

status $\mathbf{X} \geq \mathbf{0}$. In the table, a ? entry indicates the resulting status is unknown, and a blank entry represents a syntactically illegal comparison. If either status is unknown, then the resulting status is also unknown. To determine the status for more than two reaching definitions, the first two definitions are compared, and the result is then compared with each subsequent definition until all reaching definitions are considered.

Because the information stored in the definition invariant table is very similar to that in the constant table, the method of gathering the information is approximately the same. The difference is that the reaching definitions are not required to be constant. Instead, for each of the operands involved in the definition, the status of each reaching definition needs to be known. The algorithm for definition invariant propagation is given in Figure 3.

The method for determining the status of each defined variable is as follows. The number of operands involved in the assignment is determined. In the case of a simple assignment involving only one operand such as $\mathbf{X} = \mathbf{Y}$, the defined variable simply assumes the operand's status. For assignments involving two or more operands or a single operand involved in a unary operation, the status of each variable is first determined as described above. If any operand's status is unknown, then the status of the defined variable is unknown, and the next definition is considered. If the status is known, then the status of each binary or unary operation involved in the assignment is determined by using a *status decision table*, which has been defined for each arithmetic operation. The tables for addition and multiplication are shown in Table 4; the tables for the other operations (subtraction, division, and unary operators) are similar. After completing this task, the resulting status is assigned to the definition being examined.

The information stored in the definition invariant table can be used to equivalence mutants much in the same manner as for the constant table. The mutant types that can be detected are *ABS, SVR, UOI, AOR, CSR*, and *SCR*. The following conditions for each mutant type must hold:

- *ABS*: Each of the three variations *ABS*, *NEGABS*, and *ZPUSH* are different. The condition for *ABS* is that the variable's status must be $=$, $>$, or $\geq$. For *NEGABS*, the variable's status must be $=$, $<$, or $\leq$. For *ZPUSH*, the variable's status must $<$, $>$, or $\neq$.
- *SVR*: the status of the replaced variable and the status of the replacement variable must be $=$.
- *UOI*: the unary operator must be $-$, and the status of the operand must be $=$.
- *AOR*: the operator being replaced must be either $+$ or $-$, the replacement operator must be either $+$ or $-$, and the status of the second operand of the operation must be $=$.
- *SCR*: the constant replaced is 0 and the status of the replacement variable is $=$.
- *CSR*: the status of the replaced variable is $=$ and the replacement constant is 0.

The second invariant table, the statement invariant table, contains a list of conditions, one for each statement in the program, that describe what is known to be true at that statement. Whereas the definition invariant table contains information gathered solely from definitions about variables, the statement invariant table holds information gathered from definitions in addition to that gathered from other sources. These

```
Algorithm Definition Invariant Propagation
input     Program P, control flow graph cfg, and Constant Table CT (from Figure 2)
output    Status (if known) for each definition
declare   DIT []                          :  Definition Invariant Table of logical relationships
          bb                              :  basic block of P
          def                             :  definition in P
          reach_def                       :  reaching definition of a variable in P
          opnd                            :  operand
          r, temp_status, s               :  logical relationship (>, =, >, ≤, ≠, ≥)
          repeat_flag, known_flag         :  boolean

 1 DIT [d] := unknown, ∀ def in P
 2 FOREACH def in P
 3    IF (CT [d] ≠ unknown) THEN
 4       determine the relationship r between CT [d] and 0
 5       DIT [d] := r
 6 ENDFOR
 7 REPEAT
 8    repeat_flag := FALSE
 9    FOREACH bb in cfg in breadth-first order
10       FOREACH def in bb
11          known_flag := TRUE
12          FOREACH opnd in def
13             IF (opnd is not a scalar variable) OR (opnd is not a scalar constant) THEN
14                known_flag := FALSE
15             ELSE
16                IF (opnd is a scalar constant) THEN
17                   determine the relationship r between opnd and 0
18                   substitute r in the assignment expression for opnd
19                IF (opnd is a scalar variable) THEN
20                   IF (there is only one reaching definition reach_def of opnd) THEN
21                      IF (DIT [reach_def] ≠ unknown) THEN
22                         substitute the status DIT [reach_def] for opnd in the assignment expression
23                      ELSE
24                         known_flag := FALSE
25                   ELSE IF (DIT [reach_def] ≠ unknown ∀ reach_def of opnd) THEN
26                         tempstatus := DIT [reach_def_1]
27                         FOR i := 2 TO number of reaching definitions
28                            tempstatus := Definition_Status_Decision_Table[tempstatus, DIT [reach_def_i]]
29                         ENDFOR
30                         IF (tempstatus ≠ unknown) THEN
31                            substitute tempstatus for opnd in the assignment expression
32                         ELSE
33                            known_flag := FALSE
34                   ELSE
35                      known_flag := FALSE
36          ENDFOR
37          IF (known_flag = TRUE) THEN
38             FOREACH binary operation * OR unary operation ~ involved in the assignment expression
39                evaluate the resulting status s using the appropriate Operator_Status_Decision_Table
40                IF (s ≠ unknown) THEN
41                   substitute the status s in the assignment expression for the operation
42                ELSE
43                   known_flag := FALSE
44          IF (known_flag = TRUE) THEN
45             IF (DIT [d] ≠ s) THEN
46                DIT [d] := s
47                repeat_flag := TRUE
48       ENDFOR
49    ENDFOR
50 UNTIL (repeat_flag = FALSE)
```

Figure 3: Definition Invariant Propagation

	$<$	$=$	$>$	$\leq$	$\neq$	$\geq$
$<$	$<$	$<$	?	$<$	?	?
$=$	$<$	$=$	$>$	$\leq$	$\neq$	$\geq$
$>$	?	$>$	$>$	?	?	$>$
$\leq$	$<$	$\leq$	?	$\leq$	?	?
$\neq$	?	$\neq$	?	?	?	?
$\geq$	?	$\geq$	$>$	?	?	$\geq$

Addition Table

	$<$	$=$	$>$	$\leq$	$\neq$	$\geq$
$<$	$>$	$=$	$<$	$\geq$	$\neq$	$\leq$
$=$	$=$	$=$	$=$	$=$	$=$	$=$
$>$	$<$	$=$	$>$	$\leq$	$\neq$	$\geq$
$\leq$	$\geq$	$=$	$\leq$	$\geq$	?	$\leq$
$\neq$	$\neq$	$=$	$\neq$	?	$\neq$	?
$\geq$	$\leq$	$=$	$\geq$	$\leq$	?	$\geq$

Multiplication Table

Figure 4: Addition and Multiplication Decision Tables

invariants are derived and stored in terms of the data items themselves instead of the individual definitions as in the constant table and the definition invariant table.

To store these invariants and enable their propagation, two tables are used. The *statement invariant table* contains the invariants for each statement in the program. The *edge invariant table* contains the relationships known to be true whenever each edge in the program control flow graph is traversed. The algorithm for invariant propagation within these tables is given in Figure 5.

Invariants are derived from IF statements, loops, computed GOTO statements, and simple assignment statements (when the value of right hand side can be determined). Although full symbolic evaluation is not used [BEL75, Kin76], a symbolic evaluation-like algorithm is used. The algorithm for detecting invariants is given in Figure 6. After the Invariant Detection Algorithm has been executed, the Invariant Propagation Algorithm is used to propagate these invariants within the tables.

The information stored in the statement invariant table is used to detect equivalent mutants of types *ABS, SVR, UOI, CSR*, and *SCR*. For each mutant type, the following conditions must hold:

- *ABS*: Again, each of the three variations *ABS*, *NEGABS*, and *ZPUSH* are different. The condition for *ABS* is that the variable's relationship with zero must be $\geq$. For *NEGABS*, the variable's relationship with zero must be $\leq$. For *ZPUSH*, the variable's relationship with zero must be $\neq$.
- *SVR*: the relationship between the replaced variable and the replacement variable must be $=$.
- *UOI*: the unary operator must be $-$, and the second operand must be equal to zero.
- *CSR*: the replaced variable must be equal to the replacement constant.
- *SCR*: the replacement variable must be equal to the replaced variable.

3.5 Detecting Equivalent Mutants Using Common Subexpressions

Compiler optimizers recognize the common subexpressions that often arise as temporary variables during the compilation process. This research uses this technique to recognize equivalence between two variables.

Our algorithm establishes two tables. The first is the *subexpression table* in which the subexpressions recognized from the code are kept. The second is the *data item definition table*, which stores information regarding which subexpression is assigned to each data item. In other words, the first table acts as a

```
Algorithm  Invariant Propagation
input      Program P and control flow graph cfg
output     Invariants for each block and each edge in P
declare    SIT [] :  Statement Invariant Table of logical relationships indexed by statements
           EIT [] :  Edge Invariant Table of logical relationships indexed by control flow edges
           bb     :  basic block of P
           stmt   :  statement number
           edge   :  cfg_edge
           inv    :  invariant


 1  REPEAT
 2     FOREACH bb in cfg
 3        form the intersection of all sets of invariants of the incoming edges of bb
          -- only those invariants appearing on all incoming edges
 4        add these invariants to SIT [stmt], where stmt is the first statement in bb
 5     ENDFOR
 6     FOREACH bb in cfg
 7        FOREACH stmt in bb
 8           FOREACH inv in SIT [stmt]
 9              IF (stmt does not define a variable contained in inv) THEN
10                 IF (stmt is not the last statement in bb) THEN
11                    add inv to SIT [stmt + 1]
12                 ELSE
13                    add inv to EIT [n] for each outgoing edge of bb
14           ENDFOR
15        ENDFOR
16     ENDFOR
17  UNTIL (no new information is added to SIT [])
```

Figure 5: Invariant Propagation Algorithm

```
Algorithm  Invariant Detection
input      Program P and control flow graph cfg
output     Updated invariants for each block and each edge in P
declare    SIT []    :  Statement Invariant Table of logical relationships indexed by statements
           EIT []    :  Edge Invariant Table of logical relationships indexed by control flow edges
           stmt      :  statement number
           edge      :  cfg_edge
           inv, ninv :  invariants
           var       :  variable


 1 FOREACH stmt in P
 2    IF (stmt is an IF statement) THEN
 3       IF (all conditional expressions contain only scalar variables and scalar constants) THEN
 4          IF (there is only one conditional expression) THEN
 5             form the appropriate invariant inv from the expression
 6             n := edge number of the TRUE edge
 7             add inv to EIT [n]
 8             form the inverse invariant ninv from the expression
               -- example:  A > B ⇒ A ≤ B
 9             n := edge number of the FALSE edge
10             add ninv to EIT [n]
11          ELSE IF (all conditional expressions are connected exclusively by logical AND's) THEN
12                FOREACH conditional expression
13                   form the appropriate invariant inv from the expression
14                   n := edge number of the TRUE edge
15                   add inv to EIT [n]
16                ENDFOR
17          ELSE IF (all conditional expressions are connected exclusively by logical OR's) THEN
18                FOREACH conditional expression
19                   form the inverse invariant ninv from the expression
20                   n := edge number of the FALSE edge
21                   add ninv to EIT [n]
22                ENDFOR
23    ELSE IF (stmt is a computed GOTO statement) THEN
24       IF (the test expression is a scalar variable var) THEN
25          FOR i := 1 TO number of labels in GOTO list
26             form the invariant inv (var = i)
27             n := edge number of the corresponding branch to the i^th label
28             add inv to EIT [n]
29          ENDFOR
30    IF (stmt is an assignment statement) THEN
31       IF (the defined variable is scalar and there is only one operand involved in s) THEN
            -- e.g.  A := 5 or A := B
32          form the appropriate invariant inv
33          IF (stmt ends a basic block) THEN
34             add inv to EIT [edge] for each outgoing edge
35          ELSE
36             add inv to SIT [stmt] where stmt = the statement number of the next statement
37 ENDFOR
```

Figure 6: Invariant Detection Algorithm

Normal Code		Optimized Code	
T1	= B + C	T1	= B + C
T2	= T1 - D	T2	= T1 - D
A	= T2	A	= T2
T3	= B + C	X	= T2
T4	= T3 - D		
X	= T4		

Figure 7: Common Subexpression Detection Example

1	C = (2 * B) + 20
2	A = B + C − D
3	X = B + C − D
4	Z = A + 10
Δ	Z = **X** + 10

Figure 8: Common Subexpression Example

temporary variable table and the second keeps track of which temporary variables are assigned to which data items. Consider the assignment statement **A = B + C − D**. This assignment can be divided into the following statements using the temporary variables T1 and T2:

```
T1 = B + C
T2 = T1 - D
A  = T2
```

In this example, the subexpression table would contain entries for T1 and T2. The data item definition table would contain the information of T2 being assigned to the data item **A**.

One property that this algorithm ensures of the subexpression table is that each entry is unique. In other words, if a subexpression is encountered in the code that is already in the table, then a reference to the existing entry is used, and the redundant expression is not added to the table. The importance of this can be seen in the following example:

```
A = B + C - D
X = B + C - D
```

Figure 7 contains the normal statements resulting from the above assignments and an optimized version. Notice that because the expression assigned to T3 has already been computed and assigned to T1, T3 is not needed, and T1 can be used instead. This recognition leads to the observation that both **A** and **X** are assigned the same temporary variable and are equal. Thus, the invariant **A = X** can be added to the statement invariant table after the assignment statement of **X**.

Figure 10 contains the algorithm used to detect variable equivalence using common subexpression detection; it calls the Subexpression Deletion Algorithm in Figure 9. Using the results of these algorithms, the invariant **A = X** is added to the statement invariant table. With this knowledge, the *SVR* mutation given in Figure 8, which replaces **A** with **X**, can be determined to be equivalent.

The result of common subexpression detection is the determination of equality relationships between variables. This information is then encoded into *invariants* that are added to the statement invariant table.

```
Algorithm Subexpression Deletion (opnd)
input     Subexpression table SET[]
          opnd:  Operand of an expression (variable or expression number)
output    Updated SE []
declare   data_item    :  table of subexpression numbers for each scalar variable
          sub_expr     :  subexpression
          x            :  variable in program P
          expr_num     :  integer expression number


 1 FOREACH sub_expr in SE
 2    IF (sub_expr contains opnd as an operand) THEN
 3       expr_num := expression number of sub_expr (i.e., SE [expr_num] = sub_expr)
 4       delete sub_expr from SE
 5       FOREACH variable x in P
 6          IF (data_item [x] = expr_num) THEN
 7             data_item [x] := 0
 8       ENDFOR
 9       Subexpression Deletion (expr_num) -- note recursion
10 ENDFOR
```

Figure 9: Subexpression Deletion Algorithm

For this reason, this technique does not equivalence any mutants directly. Instead, the information gathered is used in conjunction with the results of invariant propagation to equivalence mutants.

3.6 Detecting Equivalent Mutants Using Loop Invariant Detection

The DO-loop replacement mutation operator alters the ranges of loops by changing the label in the DO-statement. During code optimization, code that is invariant through a loop is often moved outside of the loop, whereas mutation can move code either inside or outside of a loop. If a mutant changes the boundary of a loop such that invariant code is moved inside or outside of the loop, then that mutant is equivalent.

This technique is used to detect whether the mutation *DER*, which alters the bounds of a DO loop, is equivalent. The algorithm for determining equivalence for each *DER* mutant is given in Figure 11.

3.7 Detecting Equivalent Mutants Using Hoisting and Sinking

Hoisting and sinking involve determining whether code can be moved from a location where it will be executed several times to where it will be executed only once, or from locations where it appears more than once to one location. For example, if a statement will be executed on both branches of an *if-then-else* statement, and the statement does not depend upon any other code in the *if-then-else*, then possibly it can be either *hoisted* to just before the *if* or *sunk* to just after the *endif*. These techniques can be used to mark mutants equivalent in a way similar to loop invariants. They are applied to the *GLR* mutant, which replaces the label of a GOTO statement. The algorithm for detecting equivalence using hoisting and sinking for each *GLR* mutant is given in Figure 12.

```
Algorithm  Common Subexpression Detection
input      Program P and control flow graph cfg
output     Statement invariant table
declare    SET []      :  table of subexpressions in each block
           SIT         :  table of relations for each statement in P
           data_item   :  Data Item Definition Table of subexpression numbers for each scalar variable
           bb          :  basic block of P
           def         :  definition in P
           v, i, j     :  scalar variables
           expr        :  expression
           sub_expr    :  subexpression
           expr_num    :  integer expression number
           stmt        :  statement number


 1 FOREACH bb in cfg
 2    FOREACH v in P
 3       data_item [v] := 0
 4    ENDFOR
 5    delete all subexpressions from SET []
 6    FOREACH def in bb of a scalar variable v
 7       FOREACH expr of def
 8          IF (there is no sub_expr in SET [] such that expr ≡ sub_expr) THEN
 9             expr_num := expression number of expr
10             SET [expr_num] := expr
11          ELSE
12             expr_num := the expression number of sub_expr (i.e., SET [expr_num] = sub_expr)
13          ENDIF
14          substitute the expression number expr_num in the assignment expression for expr
15       ENDFOR
16       data_item [v] := expression number held in the assignment expression
17       FOREACH two distinct scalar variables i, j
18          IF (data_item [i] = data_item[j] ≠ 0) THEN
19             stmt := statement number of def
20             SIT [stmt] := (I = J)
21       ENDFOR
22       SubExpression Deletion (v) -- see Subexpression Deletion Algorithm in Figure 9
23    ENDFOR
24 ENDFOR
```

Figure 10: Common Subexpression Detection Algorithm

```
Algorithm Loop Invariant Detection
input     Program P and mutant to be considered
output    TRUE if mutant is equivalent, FALSE otherwise
declare   Old_Def             :  set of variables defined at old statement
          Old_Use             :  set of variables used at old statement
          New_Def             :  set of variables defined at new statement
          New_Use             :  set of variables used at new statement
          new_start, old_start,
          new_end, old_end,
          i, j                :  statement numbers
          new_label, old_label :  program labels


 1 IF (the DO loop can be determined to be executed at least once) THEN
 2    old_start := statement number of DO statement
 3    IF (new_label occurs physically before the old label) THEN
 4       old_end := statement number of new_label
 5       new_start := old_end + 1
 6       new_end := statement number of old_label
 7    ELSE
 8       old_end := statement number of old_label
 9       new_start := old_end + 1
10       new_end := statement number of new_label
11    ENDIF
12    FOR i := new_start TO new_end
13       IF (statement i contains a conditional branch to a statement j
14        such that j < new_start or j > new_end) THEN
15          RETURN (FALSE)
16    ENDFOR
17    FOR i := old_start TO old_end
18       IF (v is defined at i) THEN
19          Old_Def := Old_Def ∪ {v}
20       IF (v is used at i) THEN
21          Old_Use := Old_Use ∪ {v}
22    ENDFOR
23    FOR i := new_start TO new_end
24       IF (v is defined at i) THEN
25          New_Def := New_Def ∪ {v}
26       IF (v is used at i) THEN
27          New_Use := New_Use ∪ {v}
28    ENDFOR
29    IF ((Old_Def ∪ New_Def = ∅) and
30        (Old_Def ∪ New_Use = ∅) and
31        (Old_Use ∪ New_Def = ∅) and
32        (New_Use ∪ New_Def = ∅)) THEN
33       RETURN (TRUE)
34    ELSE
35       RETURN (FALSE)
36    ENDIF
37 ENDIF
```

Figure 11: Loop Invariant Detection Algorithm

```
Algorithm Hoisting and Sinking Detection
input     Program P and mutant to be considered
output    TRUE if mutant is equivalent, FALSE otherwise
declare   target1,
          target2,
          bbm, bbn :  basic block numbers
          stmt     :  statement number
          flag     :  boolean


 1 target1 := block number of original GOTO statement
 2 target2 := block number of mutant GOTO statement
 3 IF (the number of successor blocks of target1 > 1) THEN
 4    RETURN (FALSE)
 5 ELSE
 6    bbn := block number of successor block
 7 IF (the number of successor blocks of target2 > 1) THEN
 8    RETURN (FALSE)
 9 ELSE
10    bbm := block number of successor block
11 IF (bbn ≠ bbm) THEN
12    RETURN (FALSE)
13 flag := TRUE
14 FOR stmt := first statement in target1 TO last statement in target1
15    IF (there is a duplicate of statement i in target2) AND
16        (no operand involved in stmt is defined before the duplicate in target2) THEN
17       flag := TRUE
18    ELSE
19       flag := FALSE
20    IF (flag = FALSE) THEN
21       RETURN (FALSE)
22 ENDFOR
23 RETURN (flag)
```

Figure 12: Hoisting and Sinking Detection Algorithm

4 AN EQUIVALENCE DETECTION TOOL

The six techniques in section 3 are intended to be implemented as part of a testing tool. Thus, as a proof of concept, these techniques have been implemented within the Mothra mutation testing system. Although most of these algorithms run in time quadratic or cubic in the length of the program, the execution time of the Equalizer is insignificant relative to the times of executing the mutants and hand determining equivalence. The Equalizer is implemented in the *C* programming language and, like Mothra, works with Fortran 77 programs. Figure 13 shows the high-level design of the Equalizer. In Mothra, test programs are parsed into a postfix intermediate language called Mothra Intermediate Code (MIC) [KO91]. The Equalizer uses the MIC tables to build the basic block graph, find all definitions, and find the basic blocks that each definition reaches. This information is passed separately into each of the four optimization functions shown in Figure 13, which create tables indicating where dead code is found (Dead Code Table), which definitions have constant values (Constant Table), and what statements have invariants associated with them (Invariant Tables). The invariant tables have information from both invariant propagation and common subexpression detection.

In Mothra, each mutant is stored in a record called the Mutant Descriptor Record (MDR) that indicates the changes to the MIC necessary to create that mutant. After the dead code, constant, and invariant tables are constructed, they are passed to the function *Equiv*. *Equiv* applies each of the six techniques to the appropriate mutants in the MDR table.

The *dead code*, *constant propagation*, and *invariant propagation* functions use information stored within the respective tables and the data flow tables to decide whether each mutant is equivalent. The *loop invariants* function handles mutations that modify the range of a `DO` loop. For each mutant, the method that detects whether the mutation causes the addition or deletion of loop invariant code to or from a loop is applied. Similarly, the *hoisting and sinking* function considers each mutation that changes the target of a `GOTO` statement. If one of these detection functions indicates that the mutant is equivalent, then *Equiv* marks the mutant equivalent by changing its MDR.

5 EXPERIMENTATION WITH THE EQUALIZER

The Equalizer has been used to determine equivalent mutants in 15 Fortran 77 programs that cover a range of applications. These programs range in size from about 5 to 52 executable statements and have from about 180 to 3000 mutants. Because the speed of these equivalence procedures is infinitesimal when compared to the speed of mutation testing, the size of the programs is unimportant. Each program has also been analyzed by hand to determine the true number of equivalent mutants, and the Equalizer's effectiveness has been compared based on the percentage of equivalent mutants that it detected. The programs have been well studied, and the equivalent mutants have been analyzed by several researchers. Thus, it is anticipated that the number of mistakes is quite small. In some cases, programs were constructed to ensure that the software worked correctly; for example, a program was created that contained dead code to test that part of the system.

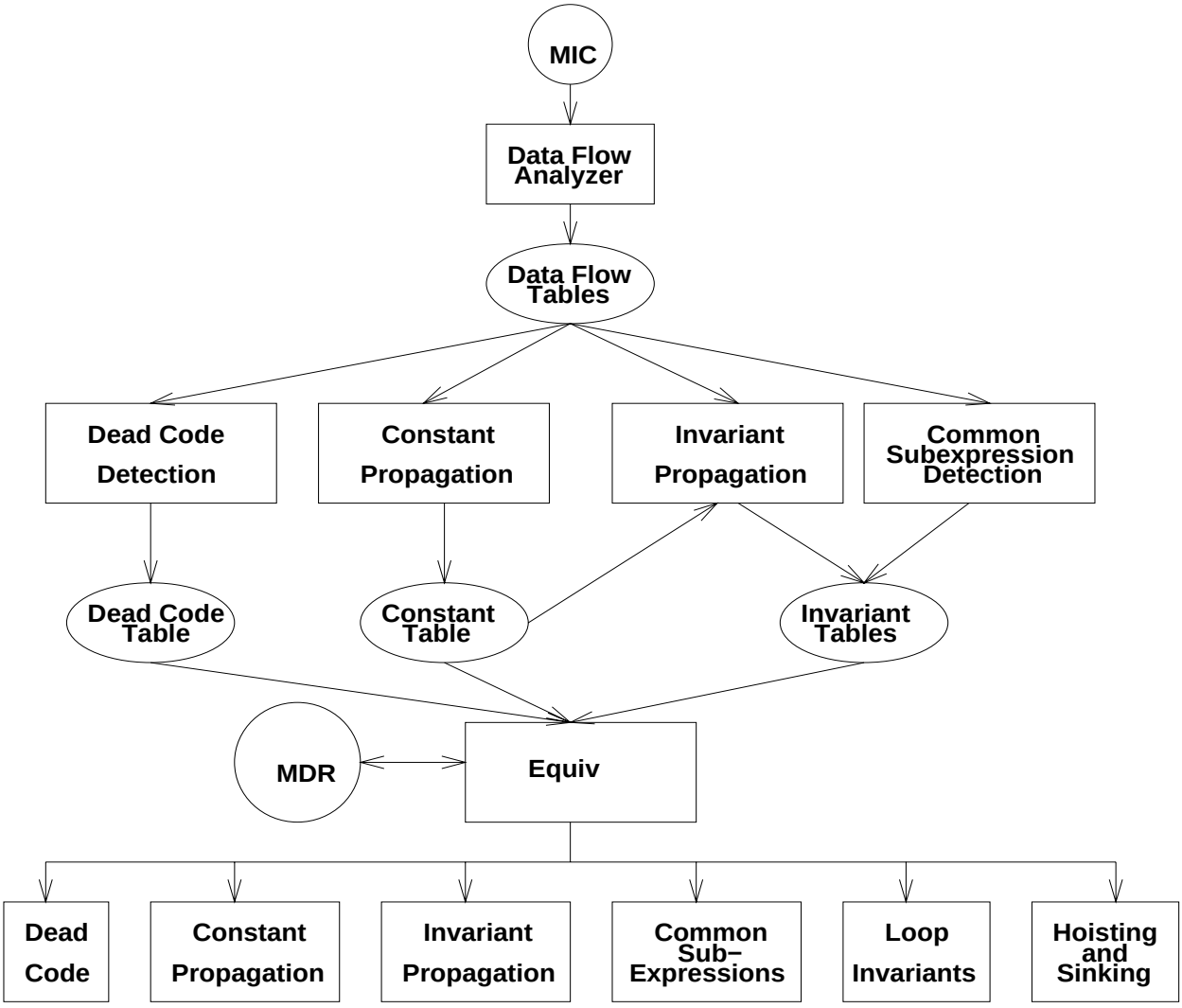

Figure 13: Flow of Data in **The Equalizer**

5.1 Equivalence Detection

The experiment used four steps:

1. For each program, each of the Equalizer's detection techniques was executed separately to count how many equivalent mutants that technique detected.

2. The mutants that were marked equivalent in step 1 were recreated (to be alive) and all detection techniques were run together to get the total number of equivalent mutants the Equalizer could detect (some equivalent mutants were detected by more than one technique).

3. The mutants that were marked equivalent in step 2 were again recreated and test cases were generated using the automatic test data generator Godzilla [DO91] and run against all mutants.

4. The remaining live mutants were analyzed for equivalence by hand to find the true number of equivalent mutants.

Program	Size (LOC)	Dead Code	Constant Propagation	Invariant Propagation	Total Detected	Total Equivalent	Percentage Detected
Bsearch	20	0	0	0	0	27	0%
Banker	48	0	1	21	21	43	49%
Bubble	11	0	5	4	5	35	14%
Cal	29	0	0	0	0	263	0%
Count	8	0	1	4	5	19	26%
Dead	8	7	0	0	7	7	100%
Deadlock	52	0	0	18	18	196	9%
Euclid	11	0	0	1	1	26	4%
Find	28	0	0	1	1	77	1%
Insert	14	0	0	10	10	48	21%
Max	5	0	1	0	1	4	25%
Mid	16	0	0	1	1	13	8%
Trismall	13	0	0	18	18	99	18%
Trityp	28	0	3	12	12	111	11%
Warshall	11	0	0	4	4	35	11%

Table 4: Equivalent Mutants Detected

The results of this experiment are displayed in Table 4. The number of equivalent mutants detected by each technique is given for each program. The techniques of loop invariants, hoisting and sinking, and common subexpression detection did not detect any equivalent mutants for these programs, and thus are not included in Table 4. The *Total Detected* column gives the number of equivalent mutants detected using all of the techniques (step 2). Because some equivalent mutants can be detected by more than one technique, the sum of the numbers of equivalent mutants detected by each technique is sometimes greater than the total number of equivalent mutants. The *Total Equivalent* column gives the total number of equivalent mutants for each program (determined in step 4). The *Percentage Detected* column gives the percentage of the total number of equivalent mutants the Equalizer detected.

One observation that can be made from the results of these experiments is that the detection power of the Equalizer depends greatly upon the program being tested. The median percentage of equivalent mutants detected was about 10%, but the standard deviation was quite high, 25%. For example, 49% of the equivalent mutants was detected for `Banker`, while only one equivalent mutant was detected for `Find`. This is largely because `FIND` contains arrays and backward `GOTO`s, which the Equalizer implementation does not handle well. `Banker` uses a well-structured algorithm with explicit loops rather than `GOTO` statements.

Each of the techniques of common subexpression detection, loop invariants, and hoisting and sinking depends on program characteristics that are relatively rare. For example, to detect an equivalent mutation using loop invariants, a labeled statement that ends a `DO`-loop must be either followed or preceded by another labeled statement, and the separating statements must be invariant in the loop. None of the subject programs had any equivalent mutants that were detectable by those three techniques, so three programs were constructed to demonstrate that the implementations of these techniques were successful and that they can detect equivalent mutants. The results of the same experiment as above for these programs are presented in Table 5. The meaning of the *Total Detected, Total Equivalent*, and *Percentage Detected* columns are the

Program	Constant Propagation	Invariant Propagation	Common SubExpr	Loop Invariant	Hoisting Sinking	Total Detected	Total Equivalent	Percentage Detected
TESTCOM	0	0	2	0	0	2	2	100%
TESTLOOP	6	4	0	1	0	7	25	28%
TESTHOIST	0	4	0	0	1	5	13	38%

Table 5: Equivalent Mutants Detected

Mutant Type	Number Equivalent	Number Detected	Percentage Detected
ABS	170	100	59%
AOR	2	0	0%
CRP	4	2	50%
CSR	3	1	33%
DER	3	0	0%
GLR	4	0	0%
LCR	6	0	0%
ROR	14	0	0%
RSR	6	1	17%
SAN	1	1	100%
SCR	5	2	40%
SDL	4	1	25%
SVR	12	5	42%
UOI	21	1	5%
Total	255	114	45%

Table 6: Equivalent Mutants by Type

same as in Table 4.

Table 6 presents the equivalent mutant distribution by type of the programs used in Table 4. Observe that the majority of the equivalent mutants for these programs are *ABS* mutants (67%). For this reason, the fact that the techniques of constant and invariant propagation detected the most equivalent mutants is not surprising because they are directly concerned with the variable's relationship with the constant zero.

6 FUTURE WORK

The current implementation treats arrays as single data items and a reference to any element of an array is treated as a reference to the entire array. For this reason, the constant and invariant propagation techniques cannot be applied to any definition containing an array reference even if the array index is known. An example of this is the statement **A(5) = 0**. From this definition, the fact that the fifth element of **A** is set to zero can be determined, and a later `use` of the fifth element would be constant. If elements of an array could be treated as individual data items, these techniques could be used to detect more information about the program being tested. Because success for this technique requires two references to the array with constant-valued indexes, it is not expected that this technique would help very often.

In the current implementation of the Equalizer, the statement invariant table consists only of simple invariants that represent relationships between two variables or between one variable and a constant. Because

the majority of the equivalent mutants detected were from invariant propagation, storing more information in the invariant tables will increase the Equalizer's ability to detect equivalent mutants. For example, if an invariant such as $\mathbf{X} > \mathbf{A} + \mathbf{B}$ was stored, and it is known that both **A** and **B** are non-negative, techniques such as those used in definition invariant propagation could be used to derive the invariant $\mathbf{X} > \mathbf{0}$. This could be further enhanced by the use of full symbolic evaluation [BEL75, Kin76].

Another potential improvement could come from further analysis of loops, perhaps using techniques developed for parallel program translation [ZC89]. Variables that are defined in loops are often *recursively* defined, that is, they are defined in terms of themselves, and the definition reaches itself as a `use`. One special case of this situation is when scalar variables are always incremented in a loop. For example, the explicitly recursive definition $\mathbf{I} = \mathbf{I} + \mathbf{1}$ can be determined to be always greater than or equal to zero if **I** is initialized to a positive value and no other definitions of **I** exist. Because this type of definition occurs frequently, this information would be helpful.

Program slicing, particularly *dynamic slicing* [KL88, AH90, ADS93], could be used to increase the effectiveness of equivalent mutant detection. Korel and Laski [KL88] introduced the notion of a dynamic program slice as "an executable part of a program whose behavior is identical to that of the original program with respect to a subset of variables of interest" during a specific execution. By restricting analysis to a program slice, the scope of information that needed to be considered for analysis could be reduced, allowing more complicated analyses to performed. It seems likely that analysis could be further improved by use of syntactic and semantic dependence as described by Podgurski and Clarke [PC90].

7 CONCLUSIONS

Mutation testing is presently very expensive, particularly in terms of the manual cost. In addition to the machine costs of executing all the mutants of a program, test cases must be generated, the output of each test case must be examined for correctness, and mutants must be analyzed for equivalence. Although progress has been made recently in automatic generation of test data [DO91, DO93], examining test case output and determining equivalent mutants are still major human costs of applying mutation testing.

This paper has made several contributions to mutation testing research. First, we have discovered how to apply the previous suggestions for automatically detecting equivalent mutants. Second, algorithms have been designed that automatically detect equivalent mutants under certain, well-specified conditions. By using techniques from data flow analysis and compiler optimization, a number of equivalent mutants can be detected automatically. Third, these algorithms have been used to implement an experimental tool to detect equivalent mutants and integrated this tool into the Mothra testing system. The Equalizer represents a partial solution to the problem of determining equivalent mutants. Fourth, this tool has been used to detect equivalent mutants in several programs. Although it is not possible to detect all equivalent mutants, the Equalizer was able to automatically detect a significant percentage, in some cases almost half. Because this problem is currently solved analytically by hand, these results can be quite useful to a software tester using mutation testing, and help make mutation testing more practical.

References

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–146, March 1976.

[Acr80] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta GA, 1980.

[ADS93] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Efficient debugging with slicing and backtracking. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 60–73, Cambridge MA, June 1993.

[AH90] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, White Plains NY, June 1990.

[All69] F. E. Allen. Program optimization. *Annual Review in Automatic Programming*, 5, 1969.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[BA82] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[BEL75] R. S. Boyer, B. Elpas, and K. N. Levitt. Select–a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, June 1975. SIGPLAN Notices, vol. 10, no. 6.

[BS79] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Research report 276, Department of Computer Science, Yale University, 1979.

[DGK+88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[DO93] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

[FL88] Charles N. Fischer and Richard J. Leblanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Company, Inc, Menlo Park, CA, 1988.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KL88] Bogdan Korel and J. Laski. Dynamic program slicing. *Information Processing Letters 29*, pages 155–163, October 1988.

[KO91] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software–Practice and Experience*, 21(7):685–718, July 1991.

[Pan94] Jie Pan. Using constraints to detect equivalent mutants. Master's thesis, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1994. (Also released as technical report ISSE-TR-94-109).

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[ZC89] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. Addison Wesley, 1989.