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.

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

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

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

[Bud80] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven CT, 1980.

[Cra89] W. M. Craft. Detecting equivalent mutants using compiler optimization techniques. Master's thesis, Department of Computer Science, Clemson University, Clemson SC, 1989. Technical Report 91-128.

[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.

[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*, January 1993. To appear.

[DSL79] R. A. DeMillo, F. G. Sayward, and R. J. Lipton. Program mutation: A new approach to program testing. In *Infotech International State of the Art Report: Program Tes*
Infotech International, 1979.

[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.

[Off88] A. J. Offutt. *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. Technical report GIT-ICS 88/28. (Also released as Purdue University Software Engineering Research Center technical report SERC-TR25-P).

Original Program	Mutant Program
`IF (X .LT. 0) THEN`	`IF (X .LT. 0) THEN`
`   Z = 0`	`   Z =0`
`ELSE`	`ELSE`
`   Z =Y / X`	`   Z =Y /` **ABS**`(X)`
`ENDIF`	`ENDIF`

Figure 7: Equivalence Detection Using Constraints

[OE88].

As an example of using constraints to detect equivalent mutants, consider the program fragment in Figure 7. The path expression to the mutated statement is $X \geq 0$, the necessity constraint for the mutant is $X < 0$, thus the complete constraint system is $X > 0 \wedge X < 0$, which is infeasible.

Another opportunity for detecting equivalent mutants comes from the path expressions created for `DO` loops. For the loop

```
DO 10 I = M, N
```

we generate the path expression constraint

$$N \geq M$$

indicating that N must be larger than M inside the loop. If a mutation within the `DO`-loop constrains N to be less than M then the constraint system is $N \geq M \wedge N < M$, which is infeasible, and the mutant is equivalent.

A simple extension to the path expression constraints generated for `DO`-loops can give more opportunity for detecting equivalent mutants. For example, if we have the loop

```
DO 10 I = 1, N
```

then the path expression constraint system

$$(I \geq 1 \wedge I \leq N)$$

is true. Although this constraint is not useful for generating test cases, it can be used to detect equivalent mutants. If a mutation constrains I to be out of this range ($I \leq 0$, or $I > N$), the constraint system is infeasible and the mutant cannot be killed.

Of course, these detection opportunities depend not only on constructing constraint systems that are infeasible, but also on the ability to detect that the system is infeasible, which is also a difficult problem. Godzilla only implements this technique in a primitive way, by considering unsolved constraints as strong "evidence" that a mutant represented by unsolved constraints is equivalent. Although a definite answer is preferable, hints of this type are certainly beneficial.

Although these results are only preliminary, most equivalent mutants seem to be represented by infeasible constraints, and most of the constraints that Godzilla cannot solve are in fact infeasible, thus it seems likely that this technique will eventually be able to detect many more equivalent mutants than the compiler optimization techniques.

such as $\mathbf{X} > \mathbf{A} + \mathbf{B}$, and we know that both $\mathbf{A}$ and $\mathbf{B}$ non-negative, techniques such as those used in definition invariant propagation could be used to derive the invariant $\mathbf{X} > \mathbf{0}$.

Another potential improvement could come from more analysis of loops. 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 $\mathbf{I}$ is initialized to a positive value and no other definitions of $\mathbf{I}$ exist. Since this type of definition occurs frequently, this information would be quite helpful.

In the data flow algorithms used in the Equalizer, arrays are treated as a single data item 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 $\mathbf{A(5)} = \mathbf{0}$. From this definition, the fact that the fifth element of $\mathbf{A}$ 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. Since success for this technique requires two references to the array with constant-valued indexes, we do not expect this technique to help very often.

7 USING CONSTRAINTS TO DETECT EQUIVALENT MUTANTS

In his dissertation [Off88], Offutt describes a method for using mathematical constraints to detect equivalent mutants. In particular, the *necessity constraints* and *path expression constraints* that are used for generat test data can be examined to determine equivalence. Necessity constraints encode conditions that a test case must meet to kill a specific mutant. For example, an `abs` mutation can only be killed if a test case causes the mutated expression to have a negative value. The necessity constraint encodes that condition. A path expression constraint for a statement encodes the conditions on a test case that will cause the statement to be executed. For example, if a statement can only be reached if $X \leq 0$, then that is part of the path expression constraint for that statement. Godzilla [DGK+88] is a tool that generates necessity and path expression constraints, combines the two to create, for each mutant, a constraint system that describes a test case to reach the mutant and then kill the mutant, then satisfies the constraint system by generating a test case that will kill the mutant a high percentage of the time.

The necessity constraints and the path expression constraints can not only be used to generate test data, but also to detect equivalent mutants. The key insight is that if the combination of a necessity constraint and its path expression constraint are infeasible, then that constraint system indicates that there are no test cases that can kill the mutant, hence the mutant cannot be killed. There are severe theoretical limitations to this technique, specifically, although Godzilla's constraints have been shown to be highly effective [DO93], the path expression constraints cannot absolutely guarantee reachability. Thus, an infeasible constraint system will not guarantee that the mutant is equivalent, but in most cases it will be. In fact, an infeasible constraint system will always represent an equivalent mutant if there are no backwards `GOTO`s in the program.

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 4: Equivalent Mutants Detected

well-structured algorithm with explicit loops rather than `GOTO` statements. Another observation is that the majority of the equivalent mutants detected by the Equalizer (67%) were `abs` mutants. This reflects the fact that the techniques of constant and invariant propagation, especially definition invariant propagation, were the most successful, since they are directly concerned with the variable's relationship with the constant zero.

Each of the techniques of common subexpression detection, loop invariants, and hoisting and sinking depend 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. Since none of our subject programs had any equivalent mutants that were detectable by those three techniques, we constructed three programs 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 4. The *Total Detected*, *Total Equivalent*, and *Percentage Detected* columns are the same as in Table

6 CONCLUSIONS AND FUTURE WORK

Although mutation testing is a technique that is demonstratably effective at finding errors, it is expensive. 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], examining test case output and determining equivalent mutants are still major human costs of applying mutation testing.

The Equalizer represents a partial solution to this problem. By utilizing techniques from data flow analysis and compiler optimization, a number of equivalent mutants can be detected automatically. Although it is not possible to detect all equivalent mutants, we were able to automatically detect a significant percentage, in some cases well over half. Since this problem is currently solved completely manually, these results are quite useful. Although more empirical work is needed (larger programs, etc.), these results are certainly encouraging. Below we discuss three extensions that could be made to the Equalizer to increase its power, and in the next section introduce a new method for detecting equivalent mutants.

6.1 Extensions to the Equalizer

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. Since the majority of the equivalent mutants detected were from invariant propagation, storing more information in the invariant tables may increase the Equalizer's ability to detect equivalent mutants. For example, if we store an invariant

Program	Dead Code	Constant Propagation	Invariant Propagation	Total Detected	Total Equivalent	Percentage Detected
BSEARCH	0	0	0	0	27	0%
BANKER	0	1	21	21	43	49%
BUBBLE	0	5	4	5	35	14%
CAL	0	0	0	0	263	0%
COUNT	0	1	4	5	19	26%
DEAD	7	0	0	7	7	100%
DEADLOCK	0	0	18	18	196	9%
EUCLID	0	0	1	1	26	4%
FIND	0	0	1	1	77	1%
INSERT	0	0	10	10	48	21%
MAX	0	1	0	1	4	25%
MID	0	0	1	1	13	8%
TRISMALL	0	0	18	18	99	18%
TRITYP	0	3	12	12	111	11%
WARSHALL	0	0	4	4	35	11%

Table 3: Equivalent Mutants Detected

5.1 Equivalence Detection

Our experiment used four steps:

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

The results of this experiment are displayed in Table 3. 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, thus are not included in Table 3. The *Total Detected* column gives the number of equivalent mutants detected using all of the techniques (step 2). 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. Since some equivalent mutants can be detected by more than one technique, the sum of the numbers of mutants detected by each technique is sometimes greater than the total number of detected mutants.

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. For example, 49% of the equivalent mutants were detected for `BANKER`, while only one was detected for `FIND`. This is largely because `FIND` contains arrays and backward `GOTO`s, which are not handled well by our data flow analysis algorithms. `BANKER` uses a

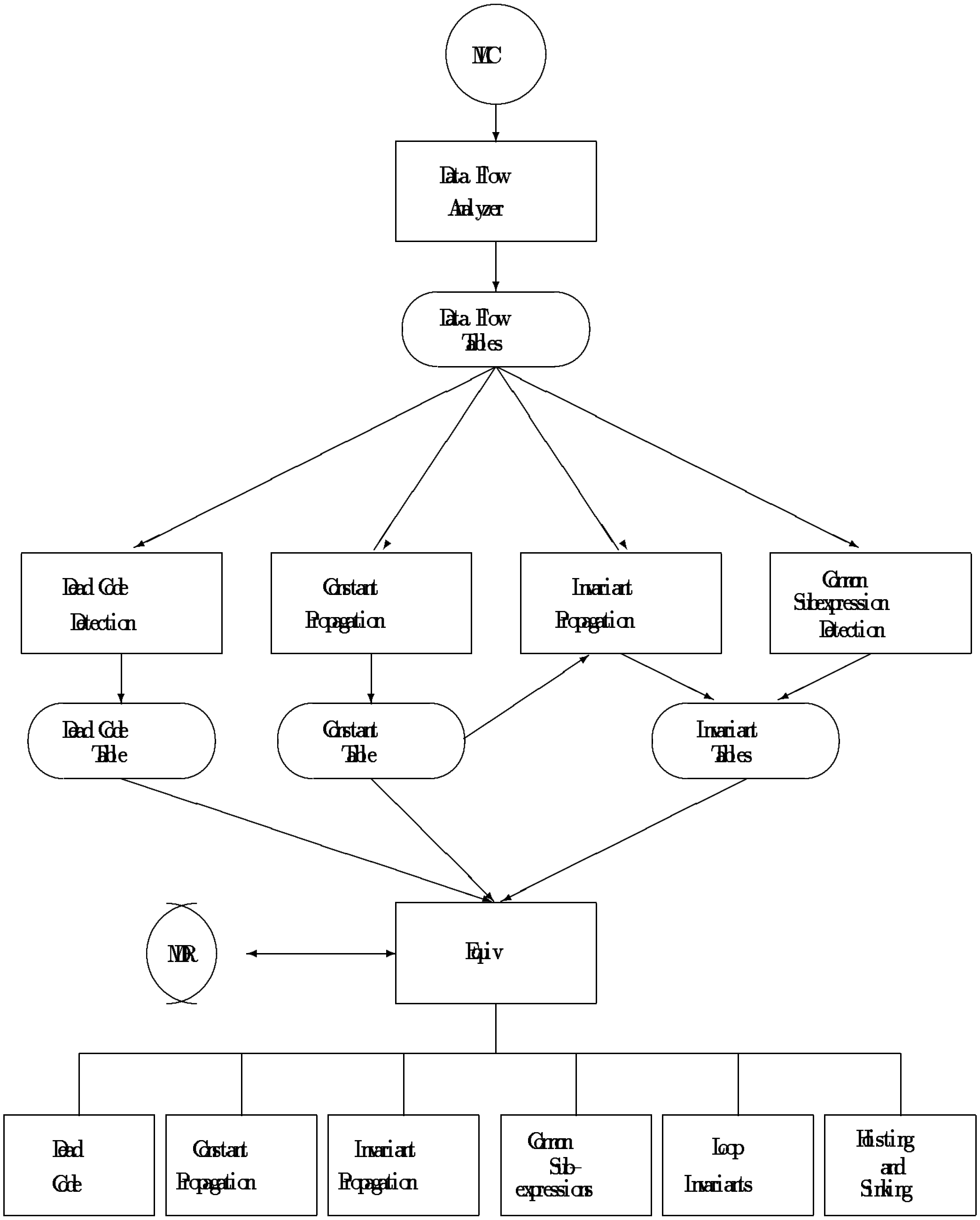

Figure 6: Flow of Data in **The Equalizer**

```
      B = 0
      IF (A .EQ. 0) GOTO 30
      A = A + 1
 30   etc.
```

Figure 5: Hoisted Version

4 AN EQUIVALENCE DETECTION TOOL

The Equalizer uses the six techniques in section 3 to automatically mark mutants equivalent in the Mothra testing system. The Equalizer is implemented in the C programming language and, like Mothra, works with Fortran 77 programs. Figure 6 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 file to build the basic block graph, find all definitions, and the basic blocks that each definition reaches. This information is passed separately into each of the four optimization functions shown in Figure 6, 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 Table). The invariant tables have information from both invariant propagation and common subexpression detection.

In Mothra, each mutant is stored in a compact 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 mutants in the MDR table.

The *dead code*, *constant propagation*, and *invariant propagation* functions use informati within the respective tables and the data flow tables to determine whether each mutant is equivalent. The *loop invariants* function considers mutations that modify the range of a `DO` loop. For each mutant, the method of detecting 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 to describe whether the mutant is equivalent. 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

We have used the Equalizer to determine equivalent mutants on 15 Fortran 77 programs that cover a range of applications. These programs range in size from about 5 to 52 executable statements and had from about 180 to 3000 mutants. We also analyzed each program by hand to determine the true number of equivalent mutants, and compared the Equalizer's effectiveness based on the percentage of equivalent mutants that it detected. In some cases, we constructed programs to ensure that the software worked correctly; for example, we created a program that contained dead code to test that part of the system.

Original Program	Mutant Program
`X = A + B`	`X = A + B`
`Y = A + B`	`Y = A + B`
`Z =` **X**	`Z =` **Y**

Figure 2: Common Subexpression Example

Original Program	Optimized Program
`DO 10 I = 1, 10`	`DO 10 I = 1, 10`
`A (I) = 0`	`A (I) = 0`
`B = 0`	`10 CONTINUE`
`10 CONTINUE`	`B = 0`

Figure 3: Loop Invariant Example

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. For example, the loop in Figure 3 contains an assignment that is moved outside of the loop during optimization. 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.

3.7 Detecting Equivalent Mutants Using Hoisting and Sinking

Hoisting and sinking is similar to loop invariants optimization. Again, it is best understood through an example. In Figure 4 is a program fragment and mutant that replaces the target of the first GOTO with the label 20.

This program fragment is a candidate for a “hoisting” optimization. The variable B is set to zero in both branches of the IF statement. A hoisting optimization would move B before the GOTO, as shown in Figure 5. Because we can do this hoisting, the mutant in Figure 4 is equivalent to the original program. As with loop invariants, if a mutation operator results in a program that could be produced by the optimization, then that mutant is equivalent.

Original Program	Mutant Program
`IF (A .EQ. 0)` **GOTO 10**	`IF (A .EQ. 0)` **GOTO 20**
`A = A + 1`	`A = A + 1`
`20 B = 0`	`20 B = 0`
`GOTO 30`	`GOTO 30`
`10 B = 0`	`10 B = 0`
`30` *etc.*	`30` *etc.*

Figure 4: Hoisting Optimization Example

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 we implemented is modeled after the procedure described by Allen [All69], however, ours propagates constants not only within block boundaries but also across these boundaries. 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 *constant table* that has one entry for each definition. If a definition is determined to be constant, then that constant 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.

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. We separate these invariants into two categories. The first group of invariants pertains to the definitions contained in the program and are stored in the *definition invariant table*. The second group is a more general group that includes invariants 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 cannot be killed when a variable 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 that are equivalent, a valuable piece of information is the relationship between a variable and the constant zero (i.e., $X > 0$). Often, even if the variable's constant value cannot be determined, its relationship with zero can, so we store that information as the *status* of the variable in the *definition status decision table*. This information is used to determine equivalent mutants when a mutant cannot be killed when a variable has the status marked in the definition status decision table. For example, to kill an `abs` mutant, the variable must have a value that is greater than zero. If the *definition status decision table* indicates the variable is strictly negative, the mutant is equivalent.

3.5 Detecting Equivalent Mutants Using Common Subexpression

Detecting equivalent mutants through common subexpression elimination can best be described through an example. Consider the program fragment and one of its mutants shown in Figure 2. Using techniques for common subexpression elimination, we can determine that X and Y have the same value when Z is defined. Thus the mutant is equivalent.

3.1 Data Flow Analysis

Data flow is a well-known program analysis technique used for compiler optimization and software testing. It is not conceptually difficult, but implementations of data flow are technically detailed and tend to be expensive to run. The terms used in this paper come from Allen and Cocke [AC76].

A variable is *defined* (a `def`) when it is assigned a value, i.e., 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` with 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 can be found 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 (e.g., 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.

3.2 Equivalencing Mutants Using Dead Code Detection

Any statement that can never be executed or whose execution is irrelevant is considered dead code. 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. 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. Such a node can easily be detected by executing a breadth-first traversal of the flow graph starting from the start node. Any node that is isolated from the start node will not be visited. 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. Any mutation that acts on a statement that has a dead definition will be equivalent.

Level	Percent of Equivalent	Percent of All Mutants
1	31.1	2.3
2	2.8	0.13
3	40.8	2.0
4	22.9	1.4
5	2.4	0.14

Table 2: Percentages of Equivalent Mutants by Level

randomly from all live mutants after test cases had been developed that eliminated enough mutants so that about half of the remaining mutants were equivalent. A *type 1* error was considered to be marking a non-equivalent mutant as equivalent, and a *type 2* error was marking an equivalent mutant non-equivalent. Type 2 errors are not serious, since the mutant remains in the system to be reconsidered.

The disturbing result of Acree's experiment was that people judged correctly only about 80% of the time. The humans made type 2 errors 12% of the time and type 1 errors 8% of the time. Since type 2 errors are “correctable” during later testing, it is really only type 1 errors that require attention. The advantage of using automated techniques to detect equivalent mutants is not that the technique would not make mistakes, but that the mistakes made would all be of type 2. An automated tool (if implemented correctly) would not convince itself that a killable mutant was equivalent.

3 COMPILER OPTIMIZATION TECHNIQUES

Baldwin and Sayward [BS79] proposed using compiler optimization strategies to detect equivalent mutants. They discussed generally how the techniques would work; we have designed algorithms (presented in Craft's thesis [Cra89]), and implemented the algorithms. The key intuition behind Baldwin and Sayward's approach is that many equivalent mutants are, in some sense, either optimizations or de-optimizations of the original program. The transformations that code optimizers make produce equivalent programs. So when an equivalent mutant satisfies a code optimization rule, algorithms can detect that the mutant is in fact equivalent. Baldwin and Sayward describe six types of compiler optimization techniques that can be used to detect equivalent mutants:

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. Because of space limitations, this is only an overview. All the details, including algorithms and complete rules for which types of equivalent mutants can be detected, can be found in Craft's thesis [Cra89]. Because these techniques depend on a data flow analysis of the program, we first present some of the basic concepts of data flow analysis.

Mutant Type	Percent of Equivalent	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

Table 1: Equivalent Mutant Percentages

Fortunately, we do have one advantage over the general equivalence problem in the context of mutation testing. Specifically, we do not have to determine the equivalence of arbitrary pairs of programs. 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.

2.1 Budd's Equivalent Mutant Difficulty Levels

Budd [Bud80] classifies equivalent mutants by how difficult it is to detect that they are equivalent. One of his observations is that equivalent mutants are not evenly distributed among the 22 mutant types. In fact, the equivalent mutants tend to cluster among only a few types. Table 1 summarizes statistics from the programs used in section 5 of this paper. The first column in the table describes a type of mutation operator and the second column gives the percentage of the total number of equivalent mutants represented by that type. The third column gives the percentage of all mutants that are equivalent of 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.

Budd divides the equivalent mutants into five levels of detection difficulty. Level 1 is the least difficult while level 5 is the most difficult to detect. Budd's analysis showed that level 1 and level 3 equivalent mutants are by far the most common. Table 2 is from Budd's dissertation [Bud80], pg. 117, and gives the percentage of each type of equivalent mutant. Many `abs` mutants are level 3, which is why there are more level 3 equivalent mutants. An encouraging aspect of Table 2 is that Budd claimed it should be possible to automatically detect equivalent mutants of type 1 through 4 — over 95% of all equivalent mutants by his count.

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. Acree chose two subjects to examine 50 mutants in each of four programs. These mutants were chosen

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 assumed to be incorrect and the mutant is killed.

After all of the test cases have been executed against all of the mutants, 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, new test cases need 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. Thus there is no need for it to remain in the system for further consideration.

1.2 Equivalent Mutants

The last mutant in Figure 1 is an equivalent mutant. Note that the reference to **I** has been replaced by a reference to **MIN**. Since these two variables always have the same value at this point in the program, 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.

The equivalent mutant in Figure 1 is easy to detect manually. However, recognizing equivalent mutants, usually done by human examination, is one of the most expensive parts of the mutation process. This paper describes algorithms to the problem of automatically detecting equivalent mutants that are based on suggestions by Baldwin and Sayward [BS79]. These algorithms have been implemented in a program that automatically detects certain equivalent mutants. In section 2, the problem is examined, and previous work done in solving this problem is presented. Six techniques for partially solving this problem involving data flow analysis and compiler optimization strategies are presented in section 3. These techniques are very involved, and the algorithms and complete rules are in Craft's thesis [Cra89]. 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, concluding remarks and suggestions for further research are presented in section 6.

2 DETECTING EQUIVALENT MUTANTS

Part of the reason that recognizing equivalent mutants is one of the most expensive mutation testing operations is that equivalent mutant detection is usually done by hand. 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 adequate test data for a program, there is also a computable procedure for checking if that program is equivalent to another program and vice versa. They also show that, in general, neither of these problems is decidable. Thus, there can be no complete algorithmic solution to the equivalence problem.

```
    FUNCTION MIN (I,J)
1   MIN = I
Δ   MIN = J
2   IF (J .LT. I) MIN = J
Δ  IF (J .GT. I) MIN = J
Δ  IF (J .LT. I) TRAP
Δ  IF (J .LT. MIN) MIN = J
3   RETURN
```

Figure 1: Function MIN

program is effectively infinite, so we must find a finite number of test cases that will give us some confidence that the program is correct.

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 divides test data into subsets that will detect the same general kinds of faults. The faults that are usually targeted are typical mistakes that programmers make. *Mutation testing* [DLS79] is one such fault-based testing method.

1.1 Mutation Testing Overview

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

Figure 1 contains a simple Fortran function with three mutated lines (preceded by the Δ symbol). Note that each of the mutated statements represents a separate program. The most recent mutation system, Mothra [DGK^{+}88], uses 22 types of mutation operators to test Fortran 77 programs. These operators have been developed and refined over 10 years through several mutation systems. The 22 mutation operators supported by the Mothra system can be divided into three general classes: *statement analysis*, *predicate and domain analysis*, and *coincidental correctness*. Statement analysis mutants check for statement coverage, statement necessity, and correct label usage. Predicate and domain analysis mutants check for cases where programmers make errors inside expressions, for example, using the wrong arithmetic operator or an incorrect comparison operator. The coincidental correctness operators check for cases where the programmer uses the wrong variable name or array reference. The first and fourth mutants in Figure 1 are *coincidental correctness* mutants, the second is a *predicate and domain analysis* mutant, and the third is a *statement analysis* mutant.

Using Compiler Optimization Techniques
to Detect Equivalent Mutants

A. Jefferson Offutt *
Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
email: ofut@gmuvax2.gmu.edu

W. Michael Craft
Department of Computer Science
Clemson University
Clemson, South Carolina

September 1992

Abstract

Mutation is a software testing technique that requires the tester to generate test da
specific, well-defined errors. Mutation testing executes many slightly differing versi
the same program to evaluate the quality of the data used to test the program. Alth
are generated and executed efficiently by automated methods, many of the mutants a
equivalent to the original program and are not useful for testing. Recognizing and elimi
mutants has traditionally been done by hand, a time-consuming and arduous task,
practical usefulness of mutation testing.

This paper presents extensions to previous work in detecting equivalent muta
present algorithms for determining several classes of equivalent mutants, and res
tation of these algorithms. These algorithms are based on data flow analysis and six
techniques. We describe each of these techniques and how they are used to detect
The design of the tool, and some experimental results using it are also presente
proach for detecting equivalent mutants that may be more powerful than the optimi
introduced.

Keywords—compiler optimizations, software testing, mutation testing, experimental software engineering

1 INTRODUCTION

Although progress in automating the testing of software has given us widely available software tools that automatically execute tests, report the results, and help perform regression testing, one of the most difficult technical problems is generating test data for unit testing—and despite much active research, the bulk of this effort is still left to the tester. The central test data generation problem is that the only way to ensure correctness is to test with all possible inputs. Unfortunately, the number of possible inputs to a given

---

*The bulk of this work was done while the authors were with Clemson University.