References


<table>
<thead>
<tr>
<th>Gilgil Diagram</th>
<th>Mint Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (X .LT. 0) THEN</td>
<td>IF (X .LT. 0) THEN</td>
</tr>
<tr>
<td>Z = 0</td>
<td>Z = 0</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>Z = Y / X</td>
<td>Z = Y / ABS(X)</td>
</tr>
<tr>
<td>ENDIF</td>
<td>ENDIF</td>
</tr>
</tbody>
</table>

Figure 7: Equivalent Relation Using Constraints

[GB].

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 maxiset constraint for the mutant is \(X < 0\) thus the complete constraint system is \(X > 0 \land X < 0\) which is infeasible.

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

\[
\text{DO 10 I = M, N}
\]

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 \(N \geq M \land N < M\) which is infeasible, and the mutant is equivalent.

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

\[
\text{DO 10 I = 1, N}
\]

then the path expression constraint system

\[(I \geq 1 \land I \leq N)\]

is true. Although this constraint is not useful for generating test cases, it can be used to detect equivalent mutants. If a mutation contains \(I\) 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 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. Gilzilla only implements this technique in a primitive way by considering unhandled constraints as strong “evidence” that a mutant represented by unhandled constraints is equivalent. Although a definite answer is preferable, lists of this type are certainly beneficial.

Although these results are only preliminary, not equivalent mutants seem to be represented by infeasible constraints, and most of the constraints that Gilzilla 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 current 0-1 quantization techniques.
such as $X > A + B$, and whenever both $A$ and $B$ are negative, techniques such as these would define \( \text{invariant propagation could be used to derive the invariant } X > 0 \).

Another potential approach could be to use analysis of loops. Variables that are defined in loops are often \textit{recursively} referred to, that is, they are defined in terms of themselves, and the definition becomes self-contained. One special case of this situation is when variables are always incremented in a loop; for example, the explicitly recursive definition $I = I + 1$ 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. Since this type of definition occurs frequently, this information would be quite helpful.

In the data flow analysis used in the Equalizer, arrays are treated as a single data item and a reference to an element of an array is treated as a reference to the entire array. In this manner, the constant and \textit{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 $A(5) = 0$. In 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 and tested. Since access for this technique requires two references to the array with constant-valued indices, we do not expect this technique to help very often.

7 \textbf{USING CONSTRAINTS TO DETECT EQUIVALENT MUTANTS}

In his dissertation \cite{Ghita1998}, Ghita describes a method for using mathematical constraints to detect equivalent mutants. In particular, the \textit{necessity constraints} and \textit{path expression constraints} that are used for generating test data can be used to detect 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 modified expression to have a negative value. 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. Ghita \cite{Ghita1998} + 8] is a tool that generates necessity and path expression constraints, and on the test case it is for, each mutant, a constraint system that describes a test case to reach the mutant and then kill the mutant, thus 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 cannot only be used to generate test data, but also to detect equivalent mutants. The insight is that if the combination of 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 Ghita’s constraints have been shown to be highly effective \cite{Ghita1998}, the path expression constraints cannot absolutely guarantee reliability. Thus, the infeasible constraint system will not guarantee that the mutant is equivalent, but in many cases it will be. In fact, an infeasible constraint system means always represent an equivalent mutant if there are no backward \textit{GOTOs} in the program.
<table>
<thead>
<tr>
<th>Program</th>
<th>Constant Propagation</th>
<th>Invariant Propagation</th>
<th>Common SubExpr</th>
<th>Loop Invariant</th>
<th>Hoisting Sinking</th>
<th>Total Detected</th>
<th>Total Equivalent</th>
<th>Percentage Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTOM</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>TESTOOP</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>25</td>
<td>28%</td>
</tr>
<tr>
<td>TESTST</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>13</td>
<td>28%</td>
</tr>
</tbody>
</table>

Table 4: Equivalent Mints Detected

well-structured algorithm with explicit loops rather than GOTO statements. Another observation is that the majority of the equivalent mints detected by the Equalizer (6%) were loop mints. 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 relation using loop invariants, a labeled statement that ends a DC-loop must be either followed or preceded by another labeled statement, and the separating statements must be invariant in the loop. Some of our subject programs had no equivalent mints that were detectable by these three techniques. We constructed three programs to demonstrate that the implementations of these techniques were successful and that they can detect equivalent mints. The results of the same program as above for these programs are presented in Table 4.

6 CONCLUSIONS AND FUTURE WORK

Although mutation testing is a technique that is demonstrably effective at finding errors, it is expensive. In addition to the human costs of creating 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 [102], ensuring 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 mutant 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 at a fraction of the cost. These results are quite promising. Although our empirical work is modest (large 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 rewarding 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 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
<table>
<thead>
<tr>
<th>Program</th>
<th>Del</th>
<th>Del</th>
<th>Insert</th>
<th>Del</th>
<th>Total</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANK</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>BANER</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>48%</td>
</tr>
<tr>
<td>BANE</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>48%</td>
</tr>
<tr>
<td>CL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>CONT</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>20%</td>
</tr>
<tr>
<td>ID</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>100%</td>
</tr>
<tr>
<td>IDK</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>KID</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>INH</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>20%</td>
</tr>
<tr>
<td>MK</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>ID</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>TSML</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>18</td>
<td>9</td>
<td>18%</td>
</tr>
<tr>
<td>TRIP</td>
<td>0</td>
<td>3</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>11%</td>
</tr>
<tr>
<td>VSNL</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>15%</td>
</tr>
</tbody>
</table>

Table 3 Equivalent Muts Detected

5.1 Equivalence Detection

Get experiment used for steps:

1. For each program each of the Evigator's detection techniques are executed separately to count how many equivalent muts each technique detected.

2. The muts that were marked equivalent in step 1 were recycled (to be alive) and all detection techniques were run together to get the total number of equivalent muts the Evigator could detect.

3. The muts that were marked equivalent in step 2 were again recycled and test cases were generated using the automatic test data generator Glib [19] and run against all muts.

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

The results of this experiment are displayed in Table 3. The number of equivalent muts detected by each technique is given for each program. The techniques of loop invariants, testing and sibling, and common subexpression detection did not detect any equivalent muts for those programs; thus are not included in Table 3. The Total Detected column gives the number of equivalent muts detected using all of the techniques (step 2). The Total Equivalent column gives the total number of equivalent muts for each program determined in step 4. The Percentage Detected column gives the percentage of the total number of equivalent muts the Evigator detected. Since some equivalent muts can be detected by one technique, the sum of the numbers of muts detected by each technique is sometimes greater than the total number of detected muts.

A general observation that can be made from the results of these experiments is that the detection power of the Evigator depends greatly upon the program being tested. For example, 48% of the equivalent muts were detected for BANKER, while only one was detected for FIND. This is largely because FIND contains traps and biased GOTOs, which are not handled well by our data flow analysis algorithm. BANKER uses a
Figure 6: How It's in The Equalizer
4 ANEQUIVALENCE DETECTION TOOL

The Equalizer uses the six techniques in section 3 to automatically rank mutants equivalent in the Milra testing system. The Equalizer is implemented in the C programming language and like Milra, works with Emran 77 programs. Figure 3 shows the high-level design of the Equalizer. In Milra, test programs are passed to a parser intermediate language called Milra Intermediate Git (MIG). The Equalizer uses the MIG file to build the basic block graph, find all definitions, and the basic blocks that each definition reaches. The information is passed separately into each of the four optimization functions shown in Figure 4, which create tables indicating where a dead code is found (Dead Git 'Bel'), which definitions have constant values (Constant 'Bel'), and which statements have invariants associated with them (Invariant 'Bel'). These tables help information flow with invariant propagation and common subexpression detection.

In Milra, each mutant is stored in a mutant record called the Mutant Description Table (MDT). The MDT indicates the changes to the MIG necessary to create that mutant. After the dead code, constant, and invariant tables are constructed, they are passed to the function Eq u i v. Eq u i v applies each of the six techniques to the mutants in the MDT.

The dead code, constant propagation, and invariant propagation functions use information 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 changes 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 determine whether the mutant is equivalent. If one of these detection functions indicates that the mutant is equivalent, then Eq u i v makes the mutant equivalent by changing its MDT.

5 EXPERIMENTATION WITH THE EQUALIZER

We used the Equalizer to determine equivalent mutants on 15 Emran 77 programs that cover a range of applications. The programs range in size from about 5 to 32 executable statements and from about 100 to 300 mutants. We also analyzed each program 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.
### 3.6 Detecting Equivalent Mutants Using Loop Invariant Detection

The DO-loop replacement mutation operator alters the range of loops by changing the label in the DO-statement. Mutating this replacement mutation can make code either inside or outside of the loop. For example, the loop in Figure 3 contains an assignment that is made outside of the loop-during quantization. If a mutant changes the boundary of a loop such that invariant code is moved inside or outside of the loop, then the mutant is equivalent.

### 3.7 Detecting Equivalent Mutants Using Hoisting and Sinking

Hoisting and sinking is similar to loop invariants quantization. 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’ quantization. The variable \( B \) is set to zero in both branches of the IF statement. A ‘sinking’ quantization would make \( B \) before the GOTO, as shown in Figure 5. Hence we can say this hoisting 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 quantization, then the mutant is 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 implemented is similar to the procedure described by Allen [Al6], however, our 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 an 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 contains 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 if a variable has the invariant related 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 invariants that are equivalent, a variable piece of information is the relationship between a variable and the constant zero (i.e., $X > 0$). 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 if a variable has the status related 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.

This 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 Gide [AC].

A variable is defined (a def) when it is assigned a value; i.e., it appears on the left handside of an assignment statement. A variable is used when it appears in the right handside 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 flow 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 nodes that immediately precede each node. This information can be obtained using a basic reachability algorithm (e.g., as given in Allen and Gide [AC]) and stored in a reach table.

After the reach table 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 any use of that variable can be determined. The information gathered about each definition in conjunction with the reach table can now be used to determine equivalent mutants.

3.2 Equivalence of Mutants Using Dead Code Detection

A 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. Ideally, 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 not used before it is referenced or is never used. 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.
mainly from all five mutants after test cases had been developed that eliminated enough mutants so that about half of the remaining mutants were equivalent. At $y p e 1$ error was considered to be making a non-equivalent mutant as equivalent, and at $y p e 2$ error was making 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 Arai’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 “unacceptable” 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 will not make mistakes, but that the mistakes made will be of type 2. An automated tool (if implemented correctly) would not ensure itself that a killed mutant was equivalent.

### Table 2: Percentages of Equivalent Mutants by Level

<table>
<thead>
<tr>
<th>Level</th>
<th>Percent of Equivalent</th>
<th>Percent of All Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31.1</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>40.8</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>22.9</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>0.4</td>
</tr>
</tbody>
</table>

3. **Compiler Optimization Techniques**

Belkin and Sigurd [89] proposed using compiler optimization strategies to detect equivalent mutants. They discussed generally how the techniques would work, with specific algorithms (presented in Glish’s thesis [68]), and implemented the algorithms. The key intuition behind Belkin and Sigurd’s approach is that any equivalent mutants are, in some sense, either optimizations or de-optimizations of the original program. The transformation that comp optimizers make produces equivalent programs. So when an equivalent mutant satisfies a comp optimization rule, an algorithm can detect that the mutant is in fact equivalent.

Belkin and Sigurd describe six types of compiler optimization techniques that can be used to detect equivalent mutants:

1. **Early-Gle-Election**
2. **Gle-Selection**
3. **Inhibit-Expansion**
4. **Glish-Shrinkage-Election**
5. **Exp-Inhibit-Selection**
6. **Hasting-and-Snaring**

These six techniques are described in the rest of this section. Due to 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 Glish’s thesis [68]. These techniques depend on a data flow analysis of the program, so next present some of the basic concepts of data flow analysis.
<table>
<thead>
<tr>
<th>Mutant Type</th>
<th>Percent of Equivalent</th>
<th>Percent of All Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Value Insertion</td>
<td>58.3</td>
<td>34.0</td>
</tr>
<tr>
<td>Solar for Constant Replacement</td>
<td>16.1</td>
<td>17.0</td>
</tr>
<tr>
<td>Array for Constant Replacement</td>
<td>11.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Array for Solar Replacement</td>
<td>39.1</td>
<td>0.19</td>
</tr>
<tr>
<td>Solar Value Replacement</td>
<td>31.1</td>
<td>0.18</td>
</tr>
<tr>
<td>Var Gnu Insertion</td>
<td>30.0</td>
<td>0.15</td>
</tr>
<tr>
<td>Relational Gnu Replacement</td>
<td>24.0</td>
<td>0.07</td>
</tr>
<tr>
<td>All Gnu Mutation</td>
<td>60.0</td>
<td>0.30</td>
</tr>
</tbody>
</table>

| Table 1: Equivalent Mutant Percentages |

Eventually we chose one mutant 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 vary merely like their original programs (IEEE and Agha [1994] refer to mutants as "neighbors" of the original program). We can take advantage of this fact to develop techniques and heuristics for detecting any of the equivalent mutants.

### 2.1 Budd's Equivalent Mutant Difficulty Levels

Budd [1994] classified 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 24 mutant types. In fact, the equivalent mutants tend to cluster among only a few types. Table 1 summarizes statistics from the program 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 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 insertion 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 divided the equivalent mutants into five levels of 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 in Budd's dissertation [1994], pg 117, and gives the percentage of each type of equivalent mutant. Only abs mutants are level 3, which is why there are more level 3 equivalent mutants. An interesting 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 a lot of effort for the testers. but Apte [1984] found that it could also prevent people from making errors in making equivalent mutants. Apte chose two subjects to examine 20 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 surviving mutant falls into one of two categories. It is either testable, but no set of test cases is sufficient to kill it. In this case, a new test case must be created. If the mutant is functionally equivalent to the original program, it is equivalent to the original program. A live equivalent mutant will always produce the same output as the original program so no test case can kill it. This is not needed 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 it has been replaced by a reference to M1. Since the 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 mutant program is always identical to that of the original.

The equivalent mutant in Figure 1 is easy to detect manually. However, recognizing equivalent mutants, usually due to human examination, is one of the most expensive parts of the mutation process. This paper describes a system to the problem of automatically detecting equivalent mutants that are based on suggestions by Baldwin and Steward [159]. 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 involving this problem is presented. Six techniques for partially solving this problem in the data flow analysis and compiler optimization strategies are presented in section 3. These techniques are very included, and the algorithms and rule sets are in Giff's thesis [Giff]. An automatic equivalent mutant detector, the Equiizer, is presented in section 4, and a description of several experiments using the Equiizer is given in section 5. Finally concluding remarks and suggestions for further research are presented in section 6.

2 Detecting Equivalent Mutants

Bit 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. Claiming oneself that a mutant is equivalent is a complicated and elaborate task that requires an in-depth analysis and understanding of the program. Baldwin and Steward [159] examine the relationship 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 deciding 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
2 IF (J .LT. I) MIN = J
3 RETURN

Figure 1 Function MIN

Programs effectively infinite, so we must find a finite number of test cases that will give us some confidence that the programs 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. Fault testing criteria divide the program's input space into subsets such that every test case in the same subset has similar properties. Thus, the program can be tested using one test case from each subset. For example, statement coverage divides program paths 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 checking 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 programs make. Mutation testing [3] 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. Here, the termination 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 need to remain in the testing process since the faults represented by that mutant have been detected.

Figure 1 contains a simple Little function with three mutated lines (preceded by the ΔMark).

Note that each of the mutated statements represents a separate program. The test result mutation system

Linda [82] uses 32 types of mutation operators to test Little 77 programs. These operators have been developed and refined over 10 years through several mutation systems. The 22 mutation operators supported by the Linda 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 programs make incorrect assignments, for example, using the wrong arithmetic operator or an incorrect comparison operator. The coincidental correctness operators check for cases where the program 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
e-mail: off@gmunvax2.gmu.edu

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

September 1992

Abstract

Mutation is a software testing technique that requires the tester to generate test data specific, well-defined errors. Mutation testing executes many slightly differing versions of the same program to evaluate the quality of the data used to test the program. Although generated and executed efficiently by automated methods, many of the mutants are equivalent to the original program and are not useful for testing. Recognizing and eliminating 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 mutants and presents algorithms for determining several classes of equivalent mutants, and representations 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 presented. A new approach for detecting equivalent mutants that may be more powerful than the optimization is introduced.

Key words - 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 Gerson University.