CHAPTER VI
INTERNAL VARIABLES

*People often say that this or that person has not yet found himself. But the self is not something that one finds. It is something one creates.*

— Thomas Szasz in "The Second Sin"

A problem that is common to constraint based testing as well as path analysis techniques is that of internal variables. A variable is *internal* to a program if it is not part of the input set of the program. Internal variables are not given values as part of a test case and the program begins execution with these variables undefined. Even though internal variables cannot be given values directly from a test case, they may still appear in path expression and necessity constraints. In order to satisfy the constraints, we need to have the internal variables take on particular values.

The internal variable problem is that of generating a test case that will cause some internal variable to take on a particular value at a particular point in the program's execution. The values of internal variables are computed within the program, but they are based on the program inputs. So even though they are not under a tester's direct control, they can be controlled indirectly.

This chapter provides an overview of the internal variable problem and suggests a theoretical and a practical way to solve the problem. Experiments to explore the importance of this problem to the effectiveness of test cases are also presented.

**Solving for Internal Variables**

One way of viewing the internal variable problem is that we wish to find a test case that will cause a variable to take on a certain value within the program. An alternate view of the problem is that given a program state (actually a partial state), we wish to work backwards in the program to find the initial state. In 1973, Gips [Gips73] developed a method of constructing the inverse of an arbitrary Turing machine. His problem was stated as: given a Turing machine $M$ that for an input tape $I$ produces output tape $O$, i.e. $M(I) = O$, construct an inverse Turing machine $M'$ such that $M'(O) = I$.

Since we can certainly model our test program as a Turing machine, this construction proves that theoretically at least we can solve the internal variable problem by inverse execution. As Gips points out in his paper, $M'$ is non-deterministic since it creates all possible input tapes to $M$ that would produce $O$. Furthermore, some of the computations of $M'$ may not terminate. The theoretical result is encouraging, however it seems unlikely that we can apply his construction directly to the internal variable problem.

A more tractable approach to solving the internal variable problem is to use data flow techniques to compute the value of an internal variable $A$ as some symbolic function of the input variables, e.g., $A = F(X)$. With this function, expressions involving the internal variable can be reduced to expressions involving input variables. This function can be approximated by using data flow analysis on the program. Data flow analysis has been widely studied and applied to a number of problems [Barr79, Clar85, Fran86, Oste74, Rapp82, Tan81]. Data flow analysis examines the flow of data through a program by finding *basic blocks* in the program. A basic block is a set of sequential statements in the program with only one entry and one exit point. The values of variables upon entry to a basic block are...
dependent on the path taken to the basic block. The values of the variables upon exit from the basic block depend on the entry values and the actions taken within the basic block. Values of internal variables can therefore be determined by following a particular sequence of basic blocks through the program and constructing a function that describes the values of the variables.

As with constructing path expressions (Chapter 4), the biggest difficulty with using data flow analysis for internal variable computation is loops. Each iteration through a loop is a separate block (or sequence of blocks). The brute force approach to applying data flow analysis to loops is called loop unwinding. When a loop is unwound, its body is duplicated once for each iteration in the loop. The difficulty with loop unwinding is in deciding beforehand how many iterations the loop will perform. Even if this can be determined, the technique is very expensive if the number of iterations is large. Luckily, this kind of brute force approach is unnecessary in most cases. Since one loop execution usually looks very much like another, we make the simplification that a loop is to be executed only once. A slightly more sophisticated approach would be to unwind the loop exactly twice to allow for special cases that may arise if the loop is executed only once.

To solve for internal variables, we first construct a table of symbolic values for the internal variables. These values are used during satisfaction to resolve the internal variables in terms of constants and input variables. The table is constructed by a symbolic walk-through of the program, much as the path expressions were computed in the algorithm in Chapter IV. For each variable, a list of possible values it may take on is stored, along with the constraints that describe under what conditions the variable will have that value. This information is used during constraint satisfaction to resolve internal variables. When an internal variable needs a value to solve a constraint, a value is chosen from the table of possible values and the associated constraints are added to the list of constraints being solved.

Below are complete descriptions of the algorithms. The first part computes the expressions for the internal variables. This algorithm looks very much like Algorithm 1 in Chapter IV. Indeed, an efficient implementation would probably combine these two algorithms. The second part resolves internal variables during constraint satisfaction.
Algorithm 7: COMPUTING INTERNAL VARIABLE EXPRESSIONS

Variables:

- $CPred$ is the current predicate.
- $Pred[s]$ contains the current predicate for each statement.
- $IVExpr[s][iv]$ contains the expressions for each internal variable. For each statement and internal variable, it contains a list of $\langle\text{value};\text{constraint}\rangle$ pairs.
- $s$ is a statement in the program $P$.
- $iv$ is an internal variable in the program $P$.

1. [Initialize]
   
   $CPred = \text{TRUE}$
   
   for each statement $s$ in $P$
   
   $Pred[s] = \text{FALSE}$
   
   for each internal variable $iv$
   
   $IVExpr[s][iv] = \text{nil}$
   
   end
   
   end

2. for each statement $s$ in $P$ in order

3. $Pred[s] = Pred[s] \cup CPred$

4. $CPred = Pred[s]$

5. if $s$ is an assignment statement then

6. [Use $s$ to form $lhs = rhs$]
   
   Add $\langle rhs; CPred \rangle$ to $IVExpr[s][lhs]$

7. if $s$ is a control flow statement then

8. Update $CPred$ according to Table 9 in Chapter IV

9. end

Figure 17. Internal Variable Expression Algorithm

The algorithm terminates at statement 9 and each $\langle\text{value};\text{constraint}\rangle$ pair in $IVExpr$ represents the conditions necessary for the internal variable to take on the value. If there are $N$ statements and $M$ internal variables in $P$, then the initialization loop will execute $N \times M$ times. The loop starting on statement 2 will execute $N$ times. The assignments on statement 3 and 4 are constant time, as are the tests on statements 5 and 7 and the update on statement 8. The table update on statement 6 adds a $\langle\text{value};\text{constraint}\rangle$ pair to the end of a linked list; a constant time operation. Thus, Algorithm 7 runs in time $N \times M$.

The next algorithm is used during constraint satisfaction to resolve internal variables. When a value is needed for an internal variable, the $IVExpr$ table that is computed in Algorithm 7 furnishes an appropriate $\langle\text{value};\text{constraint}\rangle$ pair. The value is used to resolve the internal variable and the associated constraint is added to the system of constraints being solved.
Algorithm 8: RESOLVING INTERNAL VARIABLES

Variables:
- \( C \) is the current disjunctive normal form constraint system.
- \( IVExpr \) is the table of internal variable expressions computed in Algorithm 7.
- \( iv \) is the internal variable to be resolved.
- \( value \) is the value that is returned for \( iv \).
- \( c \) is the constraint that is necessary for \( iv = value \). \( c \) is added to \( C \).
- \( s \) is the statement that the current constraint refers to.
- \( cl \) is a clause in \( C \).

1. Retrieve a \(<value:constraint> \) pair from \( IVExpr \) into \( value \) and \( c \) (see note after the algorithm)
2. for each clause \( cl \) in \( C \)
3. if \( cl \) contains \( iv \) then
4. Substitute \( value \) for \( iv \)
5. end
6. Append \( c \) to \( C \) \( (C = C \land c) \)

Figure 18. Internal Variable Resolution Algorithm

Note: There may be several \(<value:constraint> \) pairs in the table that are appropriate. Obviously the value chosen for \( iv \) must satisfy the constraints. Beyond that, a heuristic such as "choose the value with the simplest associated constraint" may be used.

The algorithm terminates at line 6 and the internal variable \( iv \) has been eliminated from the system of constraints \( C \). Line 1 implies a search of all \(<value:constraint> \) pairs in \( IVExpr \). Since a variable cannot be assigned a value more than once per program statement, if there are \( N \) statements in the program, there will be no more than \( N \) \(<value:constraint> \) pairs. This step is linear in the number of program statements.

To show how these routines can be used to solve the internal variable problem, we will look at some of the data flow expressions for BUBBLE. BUBBLE is a sorting program with two nested loops and four internal variables, \( N \), \( ITMP \), \( I \), and \( J \). The complete program with statement numbers is shown in Appendix B.

The first statement in BUBBLE assigns 5 to the variable \( N \), \( N \) is treated as a constant thereafter. So the expression for \( N \) is:

\[
N = 5: \text{TRUE} \\
N = \text{undefined}: \text{FALSE}
\]

Statement 2 is a DO loop that does an implicit test and assignment. The program tests that \( N - 1 \geq 1 \) and if true, starts executing the DO loop body by assigning the value of \( N - 1 \) to \( J \). So if we assign this predicate the name \( P \): \( 1 \):

\[
P \equiv (N - 1 \geq 1)
\]

then the expression for \( J \) is:

\[
J = N - 1: P \equiv J = \text{undefined}: \neg P
\]
Statement 3 begins another DO loop with the index variable of I. If we call this predicate P 2:

\[ P_2 = (1 \leq I) \]

then the expression for I is:

\[ I = 1; P_2 \land P_1 \]

\[ I = \text{undefined}; \neg P_2 \]

The next predicate is in statement 4 and we call it P 3:

\[ P_3 = (A(I) \leq A(I+1)) \]

Statement 5 is an assignment that is executed only if P 3 is false. The expression for ITMP is:

\[ ITMP = 1; \neg P_3 \land P_1 \land P_2 \]

\[ ITMP = \text{undefined}; P_3 \]

The calculations for the internal variable proceeds through the program in this manner. Every time a conditional is encountered, a new basic block is begun, and every time an internal variable is assigned a value, a new value is computed. These expressions are then substituted into the path expressions or necessity constraints via Algorithm 8 and used during constraint satisfaction.

**Internal Variable Experimentation**

Since the internal variable algorithms were not implemented in Godzilla, the following experiment was run to measure the increase in test case effectiveness when internal variables are reduced. The programs BUBBLE and TRITYP were used, and the constraints were generated and solved in two forms. First, clauses that contained internal variables were ignored during satisfaction. This has the effect of assuming that clauses involving internal variables were always satisfied. Next, the internal variables were reduced using an analysis such as described above in Algorithms 7 and 8. This analysis was carried out by hand. The sets of test cases generated both with internal variables and without were executed against the mutants of BUBBLE and TRITYP, using the Motha mutation system version 1.2. The results are summarized in Table 11.

<table>
<thead>
<tr>
<th></th>
<th>With IV Satisfied</th>
<th>Without IV Satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TRITYP</strong></td>
<td>Test Cases Created</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td>Mutants Killed</td>
<td>855</td>
</tr>
<tr>
<td></td>
<td>Mutants Alive</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Mutation Score</td>
<td>.99</td>
</tr>
<tr>
<td><strong>BUBBLE</strong></td>
<td>Test Cases Created</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Mutants Killed</td>
<td>302</td>
</tr>
<tr>
<td></td>
<td>Mutants Alive</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Mutation Score</td>
<td>.99</td>
</tr>
</tbody>
</table>

**Table 11. Internal Variable Experiment**

As can be seen from the data, both sets of data were extremely effective at killing mutants. For both programs, well over 95% of the mutants were killed, which indicates that both sets of test data are effective at finding errors. This means that for these programs, reducing internal variables had a
negligible effect—at a great cost. This gives at least some indication that the constraints without the clauses involving internal variables are usually strong enough to detect mutants. So expensive algorithms to automatically reduce internal variables to input variables are probably not worth the cost, except in extremely critical situations.
CHAPTER VII
DETECTING EQUIVALENT MUTANTS

The only man who behaves sensibly is my tailor; he takes my measure anew every time he sees me, whilst all the rest go on with their old measurements, expecting them to fit me.

— George Bernard Shaw

In Chapter I, it was pointed out that recognizing equivalent mutants is one of the most expensive actions in mutation analysis systems. This is partially because, like test case creation, 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.

In [Budd82], Budd and Angluin show the complete 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 is no complete automatic solution to the equivalence problem.

Fortunately, we do have one advantage over the general equivalence problem in the context of mutation analysis. Specifically, we do not have to determine the equivalence of arbitrary pairs of programs. Because of the definitions of the mutant operators, mutant programs are very much like their original program (Budd and Angluin [Budd82] describe the mutants as "neighbors" of the original program). We can take advantage of this fact to find techniques and heuristics for detecting many of the equivalent mutants. This chapter presents a new method for detecting equivalent mutants that uses structural knowledge about the programs to build infeasible constraints.

This work is not the first attempt to find ways to detect equivalent mutants. In his dissertation, Budd [Budd80a] divided equivalent mutants into five levels, based on the difficulty of detecting the equivalence. Acrey also studied equivalent mutants in his dissertation [Acrey80], and performed experiments to measure how accurately humans detect equivalent mutants. Tanaka [Tanaka81] applied data flow analysis to detect equivalent mutants and implemented an equivalent mutant detection system as part of the FORTRAN Mutation System FMS.2 [Acrey79]. Baldwin and Sayward [Baldwin79] developed a set of heuristics for detecting equivalent mutants that are based largely on compiler code optimizing techniques. These methods are surveyed in the following sections, then a new equivalent detection method that uses the constraints developed for the current work is presented and analyzed.

Budd's Equivalent Mutant Difficulty Levels

The "levels" of equivalent mutants developed by Budd [Budd80a] classify the difficulty of detecting equivalence. One of his observations is that equivalent mutants are not evenly distributed among the mutant types. In fact, the equivalent mutants tend to cluster among a few of the mutant types. Table 12 summarizes statistics from the five programs in the test suite discussed in Chapter IX. The first column in the table gives the percentage of the total equivalent mutants that the equivalent mutants of the given type represent, and the second column gives the same percentage for all the mutants.
Table 12. Equivalent Mutant Percentages

<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>54.3</td>
<td>3.40</td>
</tr>
<tr>
<td>Scalar for Constant Replacement</td>
<td>16.1</td>
<td>1.70</td>
</tr>
<tr>
<td>Array for Constant Replacement</td>
<td>11.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Array for Scalar Replacement</td>
<td>3.9</td>
<td>0.19</td>
</tr>
<tr>
<td>Scalar Variable Replacement</td>
<td>3.1</td>
<td>0.18</td>
</tr>
<tr>
<td>Unary Operator Insertion</td>
<td>3.0</td>
<td>0.15</td>
</tr>
<tr>
<td>Relational Operator Replace</td>
<td>2.4</td>
<td>0.07</td>
</tr>
<tr>
<td>All Other Mutant Operators</td>
<td>6.0</td>
<td>0.30</td>
</tr>
</tbody>
</table>

The equivalent mutants can be divided into five levels of detection difficulty. Level 1 is the least difficult while Level 5 is the most difficult to detect.

Level 1 There are two kinds of level 1 equivalent mutants. One kind occurs within the range of a DO-loop. The index variable of a DO-loop must have one of the values in the range of the DO loop limits. For example, if \( i = 1 \), 10 is the loop expression, then the value for \( i \) must be between 1 and 10 within the extent of the loop. Any mutant that can only be killed when \( i \) is either less than 1 or greater than 10. The other kind occurs on variables that define the upper bound of an adjustable array. If an array parameter to a subprogram is adjustable, then one of the other parameters must define the upper bound of the array. This parameter must always be positive, thus any mutant that can only be killed when the parameter is negative is equivalent.

Level 2 These mutants can be detected by examining the statements within the same basic block as the mutated statement. For example, if two variables have the same value within a basic block then they can be interchanged without effecting the program.

Level 3 If a variable is initialized to a non-negative value and is always incremented then it will always remain non-negative. If a mutant requires the variable to take on a non-positive value, then that mutant can be determined to be equivalent.

Level 4 These mutants can be automatically detected in theory, but in practice are quite difficult. They would require some symbolic executor system to trace a large number of paths through the program. For example, in the TRITYP program, a triangle with zero or negative length sides is illegal and the first statement in TRITYP checks for this. Many of the mutants that "bypass" this initial check (by testing the wrong variable, for example) are equivalent because the subsequent tests reveal the triangle to be illegal.

Level 5 There is a class of equivalent mutants that can only be detected through an in depth understanding of the algorithm or particular problem that the test program is solving. This is the class of equivalent mutants that are truly beyond the reach of automatic analysis.

Interestingly enough, Budd’s analysis showed that level 1 and level 3 equivalent mutants are by far the most common. Table 13 is from Budd’s dissertation [Budd80a], pg. 117, and gives the percentage of each type of equivalent mutant.
Table 13. 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>2.3</td>
</tr>
<tr>
<td>2</td>
<td>2.8</td>
<td>0.13</td>
</tr>
<tr>
<td>3</td>
<td>40.8</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>22.9</td>
<td>1.4</td>
</tr>
<tr>
<td>5</td>
<td>2.4</td>
<td>0.14</td>
</tr>
</tbody>
</table>

It is interesting to note that one mutant type, abx, accounts for over half of all equivalent mutants. Many abx mutants are level 3; so it is not surprising that there are more equivalent mutants of level 3 than of other levels. An encouraging aspect of this analysis is that it should be possible to automatically detect equivalent mutants of type 1 through 4—or over 95% of all equivalent mutants.

Detecting Equivalent Mutants By Hand

The results of an experiment conducted by Acree [Acre80] are repeated here as a motivation for automatically detecting equivalent mutants. It is obvious that detecting equivalent mutants by automatic means can save much time and energy for the testers—Acree’s results demonstrate that people also make errors in marking equivalent mutants. So if we can find ways to detect equivalent mutants automatically then we can increase the accuracy as well as the efficiency of equivalence detection.

For his experiment, Acree chose two subjects and 4 programs and had them examine 50 mutants of each program. These mutants were chosen randomly from all the mutants after test cases had been developed that eliminated enough mutants so that about half of the remaining mutants were equivalent. It has been observed that testers usually do not look for equivalent mutants until approximately this point in the testing process because of the sheer numbers of mutants.

Acree defined two types of errors that could be made in judging equivalence. An error of the first type is marking a non-equivalent mutant as equivalent, and the second type of error is marking an equivalent mutant non-equivalent. The second type of error is not serious, since the mutant remains in the system to be reconsidered during further testing.

The disturbing result of this experiment was that the humans judged correctness only about 80% of the time. Fortunately, 12% of the time the humans made errors of type 2 and made type 1 errors only 8% of the time. Since type 2 errors are easily “correctable” during later testing, it is really only type 1 errors that we need be concerned with.

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 mutant was equivalent when in fact the mutant is killable.

The mutation system EXPER [Acre79,But080] included some automatic equivalence detecting capabilities. The tester placed assertions in the program that contained data flow information. The assertions could declare that certain variables were positive or negative, equal to some constant, or equal to another variable. These assertions were used by EXPER to decide if certain mutations would be equivalent. For example, if an assertion stated that the variables M and N were equal, then some mutants that replaced M by N or N by M would be equivalent. These assertions allowed most level 1 and level 2 equivalent mutants to be detected.

Using Compiler Optimization Techniques to Detect Equivalent Mutants

Data flow analysis traces the flow of data values through variables in the program. Data flow analysis was described in Chapter 6. In [Bald79], Baldwin and Sayward suggested ways of using the
same techniques that are used in compiler optimization strategies to detect equivalent mutants. In [Tana81], Tanaka described an implementation of an automatic tool that was based on Baldwin and Sayward’s results.

The basic intuition behind this approach is to recognize 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, then a tool that knows these rules should be able to detect that the mutant is in fact equivalent. There are six basic types of compiler optimization techniques that can be used to detect equivalent mutants:

1. Constant propagation,
2. Invariant propagation,
3. Common subexpression elimination,
4. Loop invariants,
5. Hoisting and sinking,
6. Dead code.

Constant and invariant propagation can be used to determine a "current" value or range of values that a variable has at a given point in the program. For example, if a variable is assigned a positive constant value then data flow analysis can determine in what subsequent statements this value will be used. If the value is used in a subsequent statement that has an absolute value mutation, then that mutant is equivalent. The mutants that are detected with this technique are examples of Level 1 and Level 3 equivalent mutants.

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 19.

<table>
<thead>
<tr>
<th>Program</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B + C</td>
<td>A = B + C</td>
</tr>
<tr>
<td>D = B + C</td>
<td>D = B + C</td>
</tr>
<tr>
<td>X(A+E)=0</td>
<td>X(D+E)=0</td>
</tr>
</tbody>
</table>

**Figure 19. Common Subexpression Example**

Using techniques for common subexpression elimination, we can determine that A and D have the same value when the index for X is computed. Thus the mutant is equivalent. These mutants are examples of Level 2 equivalent mutants.

Mutants such as the DO-loop replacement operator can alter the range of loops. During code optimization, code that is invariant through a loop is often moved outside of the loop, whereas mutant operators can move code either inside or outside of a loop.

For example, the loop in Figure 20 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. These mutants are examples of Level 1 equivalent mutants.

Hoisting and sinking is similar to loop invariants optimization. Again, it is best understood through an example. In Figure 21 is a program fragment and mutant from Baldwin and Sayward's paper [Bald79]. The mutant is a glr mutant that replaces the target of the first GOTO with the label 2.

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Mutant Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (A.EQ.0) GOTO 1</td>
<td>IF (A.EQ.0) GOTO 2</td>
</tr>
<tr>
<td>A = A + 1</td>
<td>A = A + 1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B = 0</td>
<td>B = 0</td>
</tr>
<tr>
<td>GOTO 3</td>
<td>GOTO 3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B = 0</td>
<td>B = 0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>

Figure 21. Hoisting Optimization Example

This program fragment is also 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 \( B \) up before the GOTO:

\[
\begin{align*}
B &= 0 \\
\text{IF (A.EQ.0) GOTO 3} \\
A &= A + 1 \\
3 & \quad \text{etc.}
\end{align*}
\]

Figure 22. Hoisted Version

Because we can do this hoisting, the mutant is equivalent to the original program. As with loop invariants, if a mutant operator results in a program that could be produced by the optimization, then that mutant is equivalent. These mutants are examples of Level 2 equivalent mutants.

Dead code is the most obvious of the six compiler optimizations ways of detecting equivalent mutants. A statement is considered dead code if there is no path from the statement to an end of the program. Since these statements will have no effect on the final state of the program, any mutants on these statements will always be equivalent. These mutants are the simplest kind of Level 4 equivalent mutants.

Compiler optimization techniques can be used to detect many equivalent mutants of difficulty level 1, 2 and 3. Since this apparently represents almost 75% of all equivalent mutants then applying this technique can vastly improve a tester's efficiency when using a mutation system, and thereby improving the quality of his testing.
Using Constraints to Detect Equivalent Mutants

The necessity constraints and the path expression constraints can not only be used to kill mutants, but also to detect equivalent mutants. This result is based on the fact that when necessity constraints are conjoined with complete path expression constraints, if they are infeasible, then the necessity constraints represent equivalent mutants. If a system of constraints is infeasible, then the set of test cases that can kill the mutant is empty—implying that the mutant cannot be killed. Since we can only construct partial path expressions (as defined in Chapter IV), we cannot be sure that a system of infeasible constraints will represent an equivalent mutant. In many cases, and certainly in the case where we have no back-branches, an infeasible constraint system will represent an equivalent mutant.

As an example of the detecting equivalent mutants using constraints, consider the following program fragment:

<table>
<thead>
<tr>
<th>Program</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (X.GT.0) THEN</td>
<td>IF (X.GT.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>ENDF</td>
<td>ENDF</td>
</tr>
</tbody>
</table>

Figure 23. Equivalence Detection

Since the path expression to the mutated statement is \( X > 0 \) and the necessity constraint for an absolute value operator is \( X < 0 \), the conjunction of these constraints forms an infeasible region. This is an easy equivalence to recognize because the constraints are simple—involve only one variable and one constant.

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

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

we generate the path expression constraint

\[
N \geq M
\]

so that a mutant within the Do-loop that constrains \( N \) to be less than \( M \) forms an infeasible constraint and an equivalent mutant.

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

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

and the path expression constraint

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

we get a constraint that, while meaningless for creating test cases, can be used for detecting equivalent mutants. If a mutant constrains \( I \) to be out of this range (for example \( I = 0 \), or \( I > N \)) then the conjunction of the constraints forms an infeasible constraint.
Of course, these detection opportunities depend not only on constructing path expression constraints that when conjoined with the necessity constraints form an infeasible region, but also on the constraint satisfier's ability to detect that the region is infeasible. In Chapter V it was pointed out that it is often difficult to determine whether a set of constraints is truly infeasible or simply difficult to solve. One possibility is to apply theorem proving methods to try to show infeasibility. The approach taken in the Godzilla system is to consider unsolved constraints as strong "evidence" that the mutant represented by the constraints is equivalent. Although a definite answer is preferable, hints of this type are certainly beneficial. In Chapter VIII, an "equivalence table" is discussed that provides the tester access to the evidence about possibly equivalent mutants.

In fact, a definite answer is not always possible. Since the constraints and path expressions that are generated are not sufficient to guarantee that the mutant will be killed, constraints representing equivalent mutants will not always be infeasible. However, remember that the constraints and complete path expressions are necessary for the mutant to be killed. This means that if they are provably infeasible, then there can be no test case that kills the mutant. So using constraints to detect equivalent mutants will only make type 2 errors and will not make type 1 errors, excepting in the case where back-branches are present. In Chapter IX is an experiment that shows the effectiveness of applying constraints to detecting equivalent mutants.
CHAPTER VIII
IMPLEMENTATION OF A CONSTRAINT BASED GENERATOR

The proof is in the pudding.

— Parable

The research for this dissertation includes an implementation of the constraint based test data generation technique as a set of tools within the Mothra testing system, collectively called the Godzilla test data generator [DeMi78]. The tools serve several purposes. Primarily, they demonstrate the constraint based technique. Secondly, they form a vehicle for performing experiments to measure the effectiveness of the technique itself, the algorithms chosen, and the validity of the other results of this dissertation. Some of these experiments have been described in earlier chapters and the rest will be described in Chapter IX.

The subject of this chapter is the test data generation system itself. First the overall architecture is presented and then each of the three major tools is discussed. Then three auxiliary tools are described. The Godzilla tools were implemented in the programming language C on a DEC MicroVax II running Berkeley Unix 4.3 operating system. Godzilla contains about 15,000 lines of source and has been successfully ported to a variety of machines, including a Vax 11/780, a Pyramid, and several Sun systems.

Architecture of a Test Case Generation System

The Godzilla test data generation system is depicted in Figure 24. There are three major tools to the system, the Path Analyzer, the Constraint Generator, and the Constraint Satisfier. These are represented in the figure by boxes. These tools, which are designed as separate entities and implemented as separate programs, communicate through the files represented by the ovals. The arrows indicate the flow of data through Godzilla.

The path analyzer does a symbolic execution of the program, implementing the path expression rules described in Chapter IV. The constraint generator does a different type of symbolic execution of the program, constructing the necessity constraints as described in Chapter III. The constraint generator also combines the path expression constraints with the mutant types to create the predicate constraints, also discussed in Chapter III. The constraint satisfier pulls all three files of constraints together and creates test cases that satisfy the constraints.

These tools implement the algorithms described in earlier chapters so they will not be described in as much detail as the auxiliary tools.

Auxiliary Test Data Generation Tools

In addition to the three major tools described above, the test data generation system has several "auxiliary" tools. These implement functions that allow a tester greater control over the test data that is created. The architecture of the system with these additional tools included is shown in Figure 25.

One additional tool is actually an extension to the Mothra testing system's Fortran-77 parser. The parser can be found described in [Offu87] and [DeMi87]. In addition to parsing Fortran-77 code, the parser used in this system parses assertions embedded in the program and translates them into constraints to be used by the constraint satisfier. The constraint editor allows a tester to interactively modify the path expressions or the predicate constraints to eliminate internal variables or to simplify the constraints.
beyond the system's automated capabilities. The equivalence statistics tool gives the user access to the table of possibly equivalent mutants. Each of these tools is described in more detail below.

**Assertions**

The Mothra parser is a Fortran-77 parser that produces Mothra Intermediate Code (MIC), a postfix intermediate language. MIC is fully described in our description of the Mothra interpreter [Offu87] and the Mothra version 1.0 internal documentation [DeMi87]. The parser has been extended for test data generation to allow assertions to be placed in the program. Figure 26 shows an example of an assertion in the MAX function.

In the example, $M$ and $N$ are asserted to be unequal. This corresponds to a pre-condition on the program that is translated into a constraint. From the parser's point of view, ASSERT is a new keyword in the language and the expression is parsed just as an IF statement with the exception that the MIC instructions generated are stored in a different file from the MIC instructions. Assertions can be used to give the tester greater control over the test case creation process. He can constrain the range of certain variables or supply information about relationships between subroutine parameters (as in the above example). In Chapter IX a way of using assertions to test program specifications is discussed.

The Assertion Translator translates the MIC code produced by the parser into constraints that can be used by the constraint satisfier. The assertions are conjoined to the path expressions and necessity constraints during constraint satisfaction. The assertion in the above example will force every test case created for MAX to have values for $M$ and $N$ that are not equal.

**Constraint Editor**

Although the test data generator is designed to work automatically, there are times when it is convenient for the tester to exercise greater control over the constraints. To allow for these times, a constraint editor allows the tester to remove entire constraints or clauses within the constraints, modify constraints, or add additional constraints. This allows the tester to modify constraints in ways that are beyond the capabilities of the automated constraint handler.
Constraints are maintained in disjunctive normal form (DNF). The constraint editor allows the tester to make changes at either the constraint level or at the individual clause level. The semantics of the allowable changes keep the constraints in DNF. The changes do not ensure that the new constraint system is equivalent to the original, this is up to the tester. One place where the constraint editor might be used is to eliminate internal variables. For example, in TRITYP, the path expression for statement 10 is:

\[(l > 0) \land (j > 0) \land (k > 0) \land (\text{TRIANG} = 0)\]

From the previous three statements in the program, we know that for TRIANG to still be 0, the three input variables \(I, J\) and \(K\) must all have different values. So the constraint editor can be used to expand the clause \((\text{TRIANG} = 0)\) into three conjunctive clauses in terms of the test case variables:

\[(l > 0) \land (j > 0) \land (k > 0) \land (l \neq j) \land (j \neq k) \land (l \neq k)\]

Another use for the constraint editor is to eliminate redundant clauses. For example, this constraint overspecifies the variable \(X\):

\[(X > 4) \land (X \geq 7)\]

Clearly, if \(X\) is greater than or equal to 7, it will also be greater than 4. The constraint editor can be used to reduce this constraint to:

\[(X \geq 7)\]
an equivalent but simpler constraint. This change will have no functional effect on how the constraint is satisfied, but can improve the efficiency of the satisfaction.

The editor can also be used to break cycles in constraints. In Chapter V, Example 1 was a pair of constraints that constrained $Y$ to be the average of $X$ and $Z$:

\[(Z = Y + A) \land (Y = X + A)\]

One method of breaking such cycles suggested by Borning [Bor79] and Steele [Stee80] was to add redundant constraints. Following Example 1, two clauses can be added to make the constraint:

\[(Z = Y + A) \land (Y = X + A) \land (Z = X + B) \land (B = A + A)\]

These are several examples of how a constraint editor can be useful. In general, using the editor allows the tester more control over the test data creation process at the expense of forcing her to spend more time testing. The editor can best be applied in situations where the satisfier is failing. If the satisfier is failing to satisfy a path expression for a particular statement or group of statements, then the tester may want to investigate the problem and simplify these constraints. This is reminiscent of code optimization strategies where the advice is to not recode everything, but only those parts of the code that are executed the most. The way to hand-optimize constraints is to focus only on those expressions that cause the most difficulty for the satisfier.

Equivalence Table

In Chapter VII, using the test data constraints to detect equivalent mutants was discussed. The major observation was that if the necessity constraint for a particular mutant conjoined with the path expression for a statement forms an infeasible region, then the mutant is equivalent to the original program (excepting for back-branches). Unfortunately, detecting that a set of constraints is infeasible is undecidable [Budd82]. One conclusion was that even when the constraint satisfier cannot be sure that the constraints are not solvable, if the satisfier cannot find a solution then the constraints are "likely" to be infeasible—and the corresponding mutant is likely to be equivalent.

When a set of constraints is not satisfied, an entry is made in the equivalence table. The equivalence table stores the (indexes of the) necessity constraint that was not satisfied and the mutant that the constraint represented. The other parts of the constraint system (path expression, predicate constraint, assertion, etc.) are accessed through the necessity constraint. Since the likelihood of the mutant represented by an unsatisfied constraint being equivalent depends on the power of the satisfaction algorithm used, the table stores which satisfaction algorithm failed. The equivalence statistics program (equat) serves as an "equivalence advisor" by presenting the information in the equivalence table to the tester.

As an example consider the mutant of BSEARCH shown in Figure 27:
LOGICAL FUNCTION BSEARCH (LIST, ELEM)
INTEGER LIST (10), ELEM, LOW, HIGH, MID
1LOW = 1
2HIGH = 10
3MID = (LOW + HIGH) / 2
4IF (HIGH.LT.LOW) THEN
5BSEARCH = .FALSE.
6RETURN
7ELSE
8IF (ELEM.EQ.LIST(MID)) THEN
9BSEARCH = .TRUE.
10RETURN
11ELSE
12IF (ELEM.GT.LIST(MID)) THEN
# IF (ELEM.GE.LIST(MID)) THEN
13LOW = MID + 1
14ELSE
15HIGH = MID - 1
16ENDIF
17GOTO 10
18ENDIF
19ENDIF
RETURN

Figure 27. BSEARCH Equivalent Table Example

The mutant on statement 12 is an nof mutant that substitutes the GE operator for the GT operator. The necessity constraint for this mutant is:

\[(ELEM > LIST(MID)) \neq (ELEM \geq LIST(MID))\]

and the path expression to statement 12 is:

\[(HIGH \geq LOW) \land (ELEM \neq LIST(MID))\]

The constraint satisfaction algorithm (presented in Chapter V) could not find a solution to this system of constraints and neither could it demonstrate that the system was infeasible. So a test case was not generated for this mutant. Instead, the information described above was stored in the equivalence table. After execution of the mutants, eqstat can present these constraints in the tester. The tester can then either convince himself that the mutant is indeed equivalent or hand-generate a test case that satisfies the constraints.

In fact, the constraints above are infeasible. The necessity constraint can be rewritten as:

\[(ELEM = LIST(MID))\]

which makes the combined constraint system:

\[(HIGH \geq LOW) \land (ELEM \neq LIST(MID)) \land (ELEM = LIST(MID))\]

This is obviously infeasible since \(ELEM\) is constrained to be equal to and not equal to \(LIST(MID)\), an
obvious contradiction.

Detecting equivalent mutants is tedious and error prone [Acre80] when done by hand, and a tester welcomes any help, even if the help is in the form of suggestions of likely equivalent mutants. The example above is a case where automatic detection of the infeasible constraints is difficult (requiring at least a sophisticated algebraic manipulation system), but the infeasibility is relatively simple to show by hand inspection. In the case of the equivalence table, the tester is directed towards mutants that are at least difficult to kill. Moreover, the constraints that are generated give the tester a starting point from which to either determine equivalence or to generate a test case that does kill the mutant.
CHAPTER IX

EMPIRICAL STUDIES

If it matters not if you win or lose, why do they keep score?
— Adolph Rupp

A principle reason for implementing Godzilla was to provide a vehicle for experimenting with test data generation. A number of experiments have been performed to answer questions about the effectiveness of the technique, the quality of the solutions to the technical problems, the algorithms chosen and the quality of the implementation. Some of the results of these experiments have been presented earlier in the dissertation and will not be repeated here. Others have been alluded to and are presented here in their entirety.

There are nine experiments reported here, divided into four categories. The experiments are as follows:

I. Effectiveness of Technique
   • Adequacy Of Test Cases
   • Test Case Precision Experiment
   • Adequacy Comparisons
   • Finding Equivalent Mutants

II. Performance of Technique
   • Mutant Killing Time

III. Improvement of Technique Effectiveness
   • Internal Variable

IV. Improvement of Technique Performance
   • Efficiency of Test Cases
   • Combining Constraints
   • Improve Satisfaction

The four technique effectiveness experiments measure how well the constraint-based technique performs. These four experiments show that the constraint-based technique (as implemented by Godzilla) generates test sets that are extremely close to being mutation-adequate as well as being useful for other problems (in particular the problem of detecting equivalent mutants).

The technique performance experiment measures the time spent killing mutants by Godzilla versus the time spent hand-killing mutants. This experiment shows that not only does the automating test generation process shift some of the burden of the testing process from the tester to the computer, but also that we can achieve a significant speedup in the total time spent testing.

The technique effectiveness improvement experiment measures the importance of solving for the internal variables that are in the constraints. The experiment shows that the test cases created by ignoring
internal variable constraints were nearly as effective at killing mutants as test cases created that satisfy internal variable constraints. This indicates that solving the internal variable problem is an expense that is not necessary—at least with small programs.

The last set of experiments were concerned with improving the performance of the constraint-based technique. They show that reducing the (rather large) number of test cases produced by Godzila could significantly increase the performance without reducing the effectiveness of the test cases. Also, a faster satisfaction algorithm could greatly reduce the amount of time spent satisfying the constraints.

Next the programs used in these experiments are described. After that, each of the experiments are presented and discussed in detail.

The Program Suite

A suite of five Fortran-77 programs were chosen for these studies. The five programs were taken from the literature and chosen to represent different types of problems to exercise the generation capabilities in as wide a manner as possible. The five programs are BUBBLE, DAYS, FIND, GCD, and TRITYP. They are listed in Appendix B and described below.

BUBBLE is an array manipulation program that implements the bubble-sort algorithm to sort elements of a numeric array. DAYS is taken from Geller [Gell78] and Budd's dissertation [Budd80a] and is a function that computes the number of days between two given dates. FIND was studied by Hoare [Hoar71] and by DeMillo, Lipton and Sayward [DeMi78] and accepts an array A of integers and an index F. It returns the array with every element to the left of A(F) less than or equal to A(F) and every element to the right of A(F) greater than or equal to A(F). GCD was presented by Bradley [Brad70] and studied by Budd [Budd80a]. It computes the greatest common divisor of the input array. TRITYP has been widely studied in software testing [Clar83,DeMi78,Acre79,Rama76] and others, because of its relatively high branching factor and because of the ease of comparing results among various researchers. It is perhaps the "canonical" software testing example. TRITYP takes three integers as input that represent the relative lengths of the sides of a triangle. TRITYP classifies the triangle as equilateral, isosceles, scalene or illegal.

These programs are referred to collectively as "the test program suite". These experiments were conducted using the Mothra mutation testing system, version 1.2 running on a DEC MicroVax II. Some of these experiments are also reported in [DeMi88a].

Effectiveness of Technique

A basic question about a test data generation technique is how effective is the test data that is generated? Since constraint-based testing attempts to create mutation-adequate test data, the most telling measurement is how the data performs on a mutation system. Three different experiments are discussed below that measure the performance of this technique with the program suite. The last technique effectiveness experiment measures the ability of the constraints to detect equivalent mutants.

Adequacy Of Test Cases

Test case adequacy is the simplest and in some ways the most direct measurement of this test case generation method. The rationale behind the creation of the test cases is mutation-adequacy. Test cases are constructed precisely to score well on a mutation analysis system—to kill mutants. So the most direct measurement of performance is to calculate the mutation score of the test data. The mutation score (MS) was defined in Chapter 1 to be the ratio of dead over non-equivalent mutants. If the total number of mutants is M, the number of dead mutants is K, and the number of equivalent mutants is E, then the

---

1. The internal variable problem was discussed in Chapter VI.
The mutation score is calculated as:

$$MS = \frac{K}{(M - E)}.$$  

The mutation score is a quantitative measure of how well the test data approximates adequacy and therefore also of how well the data tests the program [DeMi86, Acre79, Budd80]. For this experiment, test cases were generated for each of the programs in the suite, all mutants were generated, and each test case was, in turn, executed against all remaining live mutants. This is the typical way that testers use Mothra. The results of this experiment are displayed in Table 14. The column labels are as used in the mutation score computation; TCs is the number of test cases, M is the number of mutants generated, K is the number of mutants killed by the test cases, E is the number of equivalent mutants, and MS is the mutation score. The last column, Time, represents the length of wall clock time in minutes and seconds that Godzilla took to generate the test cases.

<table>
<thead>
<tr>
<th></th>
<th>TCs</th>
<th>M</th>
<th>K</th>
<th>E</th>
<th>MS</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE</td>
<td>32</td>
<td>339</td>
<td>304</td>
<td>35</td>
<td>1.00</td>
<td>0:22</td>
</tr>
<tr>
<td>DAYS</td>
<td>419</td>
<td>3016</td>
<td>2624</td>
<td>139</td>
<td>.95</td>
<td>7:02</td>
</tr>
<tr>
<td>FIND</td>
<td>58</td>
<td>1029</td>
<td>953</td>
<td>75</td>
<td>.99</td>
<td>2:27</td>
</tr>
<tr>
<td>GCD</td>
<td>325</td>
<td>5063</td>
<td>4747</td>
<td>298</td>
<td>.99</td>
<td>14:24</td>
</tr>
<tr>
<td>TRITYP</td>
<td>420</td>
<td>970</td>
<td>862</td>
<td>107</td>
<td>.99</td>
<td>10:53</td>
</tr>
</tbody>
</table>

Table 14. Test Case Adequacy Results

As can be seen, the test data scored 95 percent or more on each of the five programs. Practical experience has shown that it is extremely difficult to manually create test data that scores above 95 percent on a mutation system. In fact, the author spent approximately 30 hours constructing a set of test data to kill the mutants for TRITYP. Given the test data that was produced automatically, it was the work of a few minutes to manually find test cases that killed TRITYP's remaining eight mutants.

Test Case Precision Experiment

The adequacy experiment measures the test data set as a whole. Each test case is constructed with a particular goal—to kill an individual mutant. For each test case to be effective, the test case must kill the specific mutant it was targeted for. Measuring the test cases individually eliminates the overkill effect, where some test cases not only kill their target mutants, but also kill mutants they were not targeted for and that may not have been killed by the test case that was targeted for them.

To measure the test cases individually, each test case is executed against only the mutant(s) that it was designed to kill. This is called a "test case precision experiment" because it measures how precise each test case is at killing its target mutants. The percentage of mutants killed gives an indication of how often this test data generation technique works.

If the number of mutants that were executed by some test case is $M'$ and $K$ is the total number of mutants that were killed, then the precision of a set of test cases is defined as:

---

2. These same column headings are used throughout this chapter. Additional headings will be defined when they are used.
3. When two mutants result in identical constraints, only one test case is generated, and this test case is targeted for both mutants. Ideas for expanding this notion to satisfying multiple but different constraints with the same test case are described later in this chapter, under Combining Constraints.
\[ P = \frac{K}{M'} \]

Table 15 shows the precision of the test cases of the five programs in the program test suite. The \( MS \) column is the mutation score as calculated by the number of mutants killed over the total number of non-equivalent mutants. Note that in this case each mutant is executed by at most one test case, so that these scores are expected to be lower. Because Godzilla does not solve for constraints involving internal variables (and other special cases, such as deeply nested constraints), there were many mutants that did not have a test case constructed. In effect, the system did not really try to kill these mutants.

<table>
<thead>
<tr>
<th>TCs</th>
<th>MS</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE</td>
<td>.40</td>
<td>.94</td>
</tr>
<tr>
<td>DAYS</td>
<td>.18</td>
<td>.61</td>
</tr>
<tr>
<td>FIND</td>
<td>.09</td>
<td>.78</td>
</tr>
<tr>
<td>GCD</td>
<td>.14</td>
<td>.72</td>
</tr>
<tr>
<td>TRITYP</td>
<td>.50</td>
<td>.81</td>
</tr>
</tbody>
</table>

Table 15. Effectiveness Experiment Results

The precision factors out the effect of the internal variable constraints. \( P \) is an approximation of the \( p \) quantity defined in Chapter II. \( p \) was defined as the probability that a test case that satisfies the necessity and reachability conditions will also satisfy the sufficiency condition. The number of mutants that were executed by some test case \( (M') \) is exactly the number of test cases that satisfied the constraints. Likewise, the number of mutants that were killed \( (K) \) is the same as the number of test cases that satisfied the sufficiency condition. So, \( P \) approximates \( p \) by computing the ratio of the number of test cases that satisfied the sufficiency condition over the number of test cases that satisfied the necessity and path expression constraints.

Table 15 shows us that the overkill effect is quite important. Unfortunately, it also shows a wide variation in the precision over even a relatively small number of programs. On the other hand, even the lowest precision is above 0.6, indicating that if we satisfy the same constraint with multiple test cases (as described in Chapter II), then we can achieve a high mutation score with only a few test cases per constraint. This indicates that a solution to the sufficiency problem (such as Morell's propagation conditions [Mor84] or the RELAY model [Rich88]) would result in only a marginal improvement in the overall strength of the test cases.

Adequacy Comparisons

Another important measurement of any test data generation technique is how the test data compares with data generated by other techniques. Comparisons of this nature are surprisingly rare in the literature. There seem to be two difficulties with performing these experiments: one is the lack of automated tools that implement the known testing techniques, and the other is the question of how to compare the testing techniques. Few experiments have used software that was developed and tested in a production environment [Basi87]. The practical problems of accessing and testing such software seems to have limited this type of experimentation.

Error seeding is a method in which errors are inserted into software and test data is compared on the basis of the percentage of errors found [Mill72, Glib77]. Error seeding has been criticized because there is no guarantee that the distribution of the seeded errors has any relation to the distribution of actual errors [Budd80a] and because there is no theoretical reason why the ability to find a few errors placed artificially in a program should generalize to the ability to find arbitrary errors [Hetz76].
Measuring the test data adequacy (as defined in Chapter I) is a way of comparing testing techniques that have been used in recent studies [Girg86, DeMi81, Nut84]. Adequacy not only measures the error-detecting capabilities of the test data, but also indicates how well the software will be tested using the test data [Budd80]. Relative adequacy of a test data set can be measured easily using a mutation system.

An experiment that used an earlier mutation system to measure the adequacy of five test data generation techniques was presented by DeMillo, Hocking and Merritt [DeMi81]. They used the TRITYP program. Since no automated tools were available for the techniques, test data was generated by hand to satisfy the criteria of each of the five testing techniques. This experiment was repeated using the Mocha mutation system. To ensure repeatability, the same version of TRITYP that was used in the previous study was used. This version of TRITYP is slightly different from the version used in the other experiments presented here and is listed in Appendix B as "Unstructured TRITYP".

The comparison experiment was extended to use test data generated with the Godzila implementation of the constraint-based testing technique. Table 16 gives the results of the test data executed against the mutants generated by the Mocha system.

<table>
<thead>
<tr>
<th>Technique</th>
<th>TCs</th>
<th>K</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Analysis</td>
<td>5</td>
<td>642</td>
<td>.667</td>
</tr>
<tr>
<td>Specifications Analysis</td>
<td>13</td>
<td>780</td>
<td>.811</td>
</tr>
<tr>
<td>Branch Analysis</td>
<td>9</td>
<td>749</td>
<td>.778</td>
</tr>
<tr>
<td>Minimized Domain Analysis</td>
<td>36</td>
<td>932</td>
<td>.969</td>
</tr>
<tr>
<td>Domain Analysis</td>
<td>75</td>
<td>943</td>
<td>.980</td>
</tr>
<tr>
<td>Constraint Based Testing</td>
<td>536</td>
<td>959</td>
<td>.997</td>
</tr>
</tbody>
</table>

Table 16. Summary of Comparison Results

As pointed out in the paper by DeMillo, Hocking and Merritt [DeMi81], data sets with high mutation scores may be considered superior to those with low scores. Every mutant that is left alive by a test case set represents a fault that the data could not detect. So a test set that earns a higher mutation score has demonstrated more error detection power.

As can be seen from the table, the current prototype version of this test data generation system produces results that are better than the five techniques studied in this experiment. Of course, one could argue that in comparison with domain analysis (the most effective of the other five techniques) we added 461 test cases to kill 16 mutants, surely an expensive approach. However, an examination of the 536 constraint-based test cases shows that only 46 were actually effective at killing mutants, compared with 41 effective test cases from the domain testing technique. Furthermore, since the constraint-based test set was generated by an automated tool, the number of test cases should be balanced against the labor-intensiveness of the other techniques.

Though exact time statistics were not kept for the original experiment [DeMi81], estimates by the participants indicate that it took approximately seven man-weeks to hand-generate the data and execute the data against the mutants. The automated method took two hours and 20 minutes. Moreover, this time was almost entirely computer time—the tester simply pushed the "go" button and功能 as the oracle.

Finding Equivalent Mutants

This last technique effectiveness experiment covers the detection of equivalent mutants. Detecting equivalent mutants was discussed in Chapter VII. Remember that the general problem is unsolvable, so constraints cannot be used to detect all equivalent mutants. To determine how effective the constraints can be at detecting equivalence, the constraints that were generated for the programs in the
test program suite was analyzed.

The constraints were analyzed to determine how many could be used to detect equivalent mutants. Each equivalent mutant had a constraint system associated with it. Some of these constraints were found to be infeasible and the equivalent mutant was detected. Others were infeasible, but Godzilla did not detect that they were infeasible. As discussed in Chapter IV, Godzilla uses partial path expressions, creating the possibility of infeasible constraints that do not represent equivalent mutants. Despite the fact that several of the experimental programs used DO-loops and both FIND and GCD contained back-branches, there were no infeasible constraint systems that did not represent equivalent mutants. Still other constraints were feasible—just not strong enough to detect the mutant to be equivalent. The following describes this more precisely. The term "equivalent constraints" is used informally to indicate a constraint system that was derived from an equivalent mutant. If the number of constraints for a program is \( C \), then this sum can be divided in several ways:

- \( C = c_U + c_E \), where \( c_U \) is the number of unsatisfied constraints and \( c_E \) is the number of satisfied constraints.
- \( C = c_E + c_K \), where \( c_E \) is the number of equivalent constraints and \( c_K \) is the number of non-equivalent constraints (killable).
- \( C = c_I + c_F \), where \( c_I \) is the number of infeasible constraints and \( c_F \) is the number of feasible constraints.

\( c_U \) and \( c_E \) can be further subdivided:

- \( c_U = c_{U_k} + c_{U_k} \), where \( c_{U_k} \) is the number of unsatisfied, equivalent constraints and \( c_{U_k} \) is the number of unsatisfied, non-equivalent constraints.
- \( c_E = c_{E_k} + c_{E_k} \), where \( c_{E_k} \) is the number of satisfied, equivalent constraints and \( c_{E_k} \) is the number of satisfied, non-equivalent constraints.

Given these quantities, we can define the unsatisfied coefficient (\( U \)) and the infeasible coefficient (\( I \)) of a set of constraints:

\[
U = \frac{c_{U_k}}{c_E}, \quad I = \frac{c_I}{c_E}
\]

These coefficients indicate the "potential" the constraints have for detecting equivalent mutants. The unsatisfied coefficient represents the equivalent mutants that had constraints that were somehow difficult to satisfy—the region they described were constrained in such a way that test cases were not found and, we assume, are difficult to find. The infeasible coefficient is a stronger measure of the potential the constraints have for detecting equivalent mutants. \( I \) represents the equivalent mutants that had constraints that could never be satisfied by any satisfaction algorithm. Godzilla did not always detect that these constraints were infeasible, but a more sophisticated system might.

Table 17 contains the calculations of each of these measurements over each program in the test suite. The "Detected" column is the ratio of detected infeasible constraints over equivalent mutants. These mutants were represented by constraints that the system was able to show were infeasible. The unsatisfied constraints were analyzed for feasibility by hand.
<table>
<thead>
<tr>
<th></th>
<th>Detected</th>
<th>$U$</th>
<th>$I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE</td>
<td>0.00</td>
<td>0.85</td>
<td>0.20</td>
</tr>
<tr>
<td>DAYS</td>
<td>0.17</td>
<td>0.73</td>
<td>0.35</td>
</tr>
<tr>
<td>FIND</td>
<td>0.00</td>
<td>0.88</td>
<td>0.25</td>
</tr>
<tr>
<td>GCD</td>
<td>0.07</td>
<td>0.84</td>
<td>0.24</td>
</tr>
<tr>
<td>TRITYP</td>
<td>0.45</td>
<td>0.49</td>
<td>0.46</td>
</tr>
</tbody>
</table>

Table 17. Equivalent Mutants Coefficients

As can be seen from this table, there is a potential for detecting a large fraction of equivalent mutants. If our satisfaction algorithm could always detect infeasible constraints, then the ratio of detected mutants would rise to the values for $I$—between 20% and 50% of the equivalent mutants. If we theorized that all unsatisfied constraints represented equivalent mutants, then $U$ tells us that on four of the five programs, we would be correct 70% to 85% of the time. This is certainly enough evidence so that it is worth beginning our search for equivalent mutants from among the unsatisfied constraints.

The satisfaction technique used in the implementation did not include any special abilities for detecting infeasible constraints. One way to do that would be to include algebraic manipulations and basic theorem proving capabilities. Moreover, the necessity constraints and path expression were not explicitly modified to attempt to construct detectably infeasible constraints. It is difficult to determine how much payoff could be gained from extending the constraints specifically for equivalence detection, but we could expect some benefits.

**Performance of Technique**

For any type of automated technique, the performance is always of high importance. Not only must the technique achieve the desired results, but it also must be able to achieve these results in a reasonable length of time. Exactly what a "reasonable length of time" means depends on the application. For example, a student in a programming class who writes a program that will be discarded after the assignment is completed might not be willing to spend as much time testing as a programmer of an automatic pilot system who knows that people's lives may depend on her software. Test data generation has long been an expensive process that has been very time-consuming as well as very difficult to automate [DeMi87a, Clar76, GeIt78, Ram676]. Because of this lack of automated tools, it is difficult to compare the technique described here with other automated methods. In this experiment, constraint based testing is compared against hand generation of adequate test sets.

**Mutant Killing Time**

The original goal of this dissertation was to automate the process of generating test data within a mutation system. The most direct way to measure its performance is in terms of improvement over the previous method—which was manual entry of test data. To do this, test data was generated with the Godzila system and executed by Mothra. The machine time, human time, and mutation score were recorded. Next, the author attempted to construct test data to reach the same mutation score, keeping track of the same times. Since one element of the measurement is the tester, wall clock time was used and the test cases were executed on an unloaded Microvax II.

The hand-generation portion of this experiment was done by the author. The author is very familiar with mutation analysis as well as each of the programs and has had much experience generating test cases for mutation analysis systems. This experiment was performed for each of the five programs in the test data generation suite.

Table 18 presents the times to generate and execute test cases by hand and by Godzila. All times are in wall clock minutes. The "Human Time" column gives the number of minutes used by the tester to create the test data, and the "Machine Time" is the number of minutes used by the computer to
execute all test cases and, for Godzilla, to generate the test cases. The "Machine Time" under the AUTHOR heading represents the mutant execution time and the "Human Time" under the GODZILLA heading represents overhead associated with generating the test cases.

The "Oracle Time" is an estimate of the time spent verifying the output of the test cases on the original program. The tester served as the oracle and the time was measured by using a stopwatch to time the examination of the output for 25 test cases per program to estimate an average time for verifying each test case. The average time spent verifying each test case was then multiplied by the number of test cases to get the total time used by the oracle. Though this is only a rough estimate, the fact that the same average time per test case was used for both the automatically generated and hand-generated test sets eliminates most of the inaccuracy. The point is that each additional test case requires more work from the human to serve as oracle.

<table>
<thead>
<tr>
<th></th>
<th>AUTHOR</th>
<th>GODZILLA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number TCs</td>
<td>Human Time</td>
</tr>
<tr>
<td>BUBBLE</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>DAYS</td>
<td>52</td>
<td>503</td>
</tr>
<tr>
<td>FIND</td>
<td>25</td>
<td>1328</td>
</tr>
<tr>
<td>GCD</td>
<td>87</td>
<td>6835</td>
</tr>
<tr>
<td>TRITYP</td>
<td>56</td>
<td>609</td>
</tr>
</tbody>
</table>

Table 18. Test Data Generation Times

As can be seen, test data automatically is constant (indeed, practically negligible), whereas the human time spent analyzing the program's mutants and developing test data increased dramatically as the programs got larger and more complicated. The machine time used during automatic generation was around twice that of the time used during human generation. The time spent executing mutants was the dominating factor, rather than the time spent generating test cases.

The speedup ($) of generating the data automatically can be defined as the time used by human generation over the total time used by automatic generation. Table 19 shows the speedups from the data in Table 18:

<table>
<thead>
<tr>
<th></th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE</td>
<td>2.00</td>
</tr>
<tr>
<td>DAYS</td>
<td>1.26</td>
</tr>
<tr>
<td>FIND</td>
<td>39.31</td>
</tr>
<tr>
<td>GCD</td>
<td>10.75</td>
</tr>
<tr>
<td>TRITYP</td>
<td>3.32</td>
</tr>
</tbody>
</table>

Table 19. Speedup of Automated Method

The values of the speedup for the programs ranged from 1.26 to 39.31. This is a large variation but it is not surprising that less speedup was achieved when it took more time to verify correctness of the output. Although this study does not tell us conclusively how much human effort will be saved by using this technique to automatically generate test data, we can certainly conclude that constraint based testing does save a significant amount of time, effort and expense.
Improvement of Technique Effectiveness

The next two experiments were designed to investigate ways in which the constraint-based technique could be improved. The effectiveness studies discussed above give some insights into how to improve the technique. For example, the precision experiment demonstrated that solving the sufficiency problem could in some cases increase the effectiveness of individual test cases.

Two more possibilities of improving the technique effectiveness are to increase the number of constraints that can be satisfied and to solve for internal variables. The next experiment quantifies the amount of improvement we can expect from these changes.

Internal Variable

One of the simplifications made when implementing Godzilla was to ignore internal variables in the necessity and path expression constraints. Since an obvious improvement for this tool would be to solve for the internal variable constraints, a study was made to determine how much of an effect these internal variable constraints had on the strength of the test data. This was done by hand-solving the internal variable constraints of BUBBLE and TRITYP in terms of the input variables using the path editor discussed in Chapter VIII.

The test data strength was compared both by executing all test cases against all live mutants and by executing each test case against only its target mutant (as in the precision experiment). The comparison is shown in Table 20. In Table 20, “Full” is the experiment with test cases executed against all live mutants and “Precision” is the experiment with test cases executed against their target mutants.

<table>
<thead>
<tr>
<th></th>
<th>IV Solved</th>
<th></th>
<th></th>
<th>IV Not Solved</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TCs</td>
<td>Killed</td>
<td>MS</td>
<td>TCs</td>
<td>Killed</td>
<td>MS</td>
</tr>
<tr>
<td>TRITYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full</td>
<td>420</td>
<td>862</td>
<td>.99</td>
<td>447</td>
<td>844</td>
<td>.98</td>
</tr>
<tr>
<td>Precision</td>
<td>420</td>
<td>428</td>
<td>.81</td>
<td>447</td>
<td>278</td>
<td>.53</td>
</tr>
<tr>
<td>BUBBLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full</td>
<td>32</td>
<td>304</td>
<td>1.00</td>
<td>32</td>
<td>304</td>
<td>1.00</td>
</tr>
<tr>
<td>Precision</td>
<td>32</td>
<td>122</td>
<td>.40</td>
<td>29</td>
<td>88</td>
<td>.29</td>
</tr>
</tbody>
</table>

Table 20. Internal Variable Comparison

As can be seen, when executing all the test cases against all the mutants (the customary way mutation is used), not solving the internal variable constraints had very little effect. The mutation scores were only decreased by 0.01 and by 0.0. On the other hand, when run individually, the test cases without the internal variables satisfied were quite a bit less powerful. This implies that the makes up for lack of solutions to the internal variables. In geometric terms, this may imply that the internal variable constraints do not significantly change the size of the effective test case set. So the internal variable problem will probably not be important until we find ways to reduce the number of test cases.

Improvement of Technique Performance

There are several ways to reduce the total number of test cases that are created by the constraint technique. Some of these have been investigated in terms of how much efficiency can be gained versus how much testing strength is lost. The last experiment presented below explores how much effort is wasted in the satisfaction algorithm used by the Godzilla implementation.

Efficiency of Test Cases

In Chapter II, an effective test case was defined as a test case that killed its target mutant. The term overkill was used earlier in this chapter to describe situations where one test case kills multiple mutants. Both these terms are related to how efficient the test case set is at killing mutants. In his dissertation, Acree [Acre80] defined an efficiency coefficient to measure the efficiency of mutant
operators. An efficiency coefficient value of near zero for a mutant operator indicates that the mutations produced from that operator were almost always being killed by test cases that were also killing mutants of other types. That is, the operator was not requiring the addition of any test cases that would not have been added for other mutant operators—the operator was redundant. This is in some sense the inverse of test case efficiency. Whereas Acree was looking for redundant mutant types, the next two experiments were designed to look for redundant test cases.

The test case efficiency is developed in terms of ideal efficiency. Then we show how to calculate the actual efficiency when using one method for reducing redundant mutants, and finally compute an estimate of an expected efficiency. Let $T_m$ be the size of a minimal set of test cases that achieve adequacy. As before, the number of mutants is $M$ and the number of mutants that were killed by the test cases is $K$. The average number of mutants killed per test case is given by:

$$V = \frac{K}{T_m}$$

So $V$ is the average number of mutants killed by a minimal set of test cases. The minimal test case efficiency ($\Theta_m$) is the ratio of the average number of mutants killed over the total number of mutants:

$$\Theta_m = \frac{V}{M} = \frac{K}{T_m \cdot M}$$

Note that $\Theta_m$ can take on a value between 0 and 1, and it approaches 1 as the number of test cases needed approaches 1 and the number of mutants killed approaches the total number of mutants.

It is very difficult to determine a theoretical value for $T_m$. But this efficiency measure can be used to determine the efficiency of the current set of test cases and the expected gain under conditions that may decrease the number of test cases.

If the number of test cases is $TC$, the efficiency of a set of test cases can be computed by:

$$\Theta_c = \frac{K}{T \cdot M}$$

Note that $\Theta_c$ takes on the same values as $\Theta_m$.

As one way of approximating an efficiency increase, we might think of eliminating test cases that do not actually kill any mutants. There are two reasons why a test case will not kill its target mutant when executed by a mutation system. First, it may be ineffective for some reason. It may execute the mutant but not be strong enough to actually kill the mutant. Secondly, by the time the mutation system executes the test case, its target mutant may already be dead. The mutant was killed by a previous test case (recall that dead mutants are not executed against subsequent test cases) due to the overkill effect.

Let $T_e$ be the number of effective test cases. The expected efficiency of this effective set of test cases would be given by substituting $T_e$ into the equation for $\Theta_m$:

$$\Theta_e = \frac{K}{T_e \cdot M}$$

Note that $T_e$ does not take into account the ordering of the execution of the test cases. That is, if the test cases were executed against the mutants in a different order, some test cases might kill mutants that were previously killed in the original order, and other test cases might not kill mutants that they killed in the original order. Thus $\Theta_e$ is one of many expected efficiency measurements. Table 21 gives values for $\Theta_c$, $\Theta_e$, and $\Theta_e$ for the five Fortran programs,
Table 21. Test Cases Efficiency

<table>
<thead>
<tr>
<th></th>
<th>TC</th>
<th>Tp</th>
<th>( \Theta_p )</th>
<th>( \Theta_e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE</td>
<td>32</td>
<td>6</td>
<td>.028</td>
<td>.149</td>
</tr>
<tr>
<td>DAYS</td>
<td>419</td>
<td>27</td>
<td>.002</td>
<td>.032</td>
</tr>
<tr>
<td>FIND</td>
<td>58</td>
<td>17</td>
<td>.016</td>
<td>.054</td>
</tr>
<tr>
<td>GCD</td>
<td>325</td>
<td>53</td>
<td>.003</td>
<td>.018</td>
</tr>
<tr>
<td>TRITYP</td>
<td>420</td>
<td>47</td>
<td>.002</td>
<td>.019</td>
</tr>
</tbody>
</table>

The inverse of \( \Theta_e \) indicates how many test cases we would need to generate if we could generate only effective test cases. Because four of the five programs are in the same 2 to 5 percent range, we might estimate that 20 to 50 test cases should be enough to kill most mutants. The test case efficiency for BUBBLE, of course, is much higher than the others. This is probably an anomaly due to the small number of mutants and test cases that BUBBLE has.

Combining Constraints

The test case efficiency coefficient showed that with proper optimizations, we could expect a sizable decrease in the number of test cases generated without losing strength in the test case effectiveness. This implies that there is some overlap in the effectiveness of the test cases. The experiment to measure this overlap combines the test case constraints. The idea is that many constraints describe overlapping sets of test cases, so that a single test case can be used to satisfy multiple constraints. To measure this, we attempt to satisfy multiple constraints with one test case, then measure the adequacy of the resulting set of test cases through mutation analysis.

An option to perform this combination was implemented in Godzilla (see the "-o" option in the `consat` man page in Appendix B). All constraints for a program are stored in a table and the constraint satisfier generates test cases for each constraint in turn—looping through the table. Godzilla was modified so that when a test case was generated, each subsequent constraint in the table was checked. If the test case also satisfied other constraints, those constraints were not satisfied later. This undoubtedly does not generate the smallest set of test cases, but it does significantly reduce the number of test cases required as well as being algorithmically simple and requiring no algebraic manipulation of the test cases.

Table 22. Combining Constraints Results

<table>
<thead>
<tr>
<th></th>
<th>Full</th>
<th>Combine</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TCs</td>
<td>Killed</td>
<td>MS</td>
<td>TCs</td>
</tr>
<tr>
<td>BUBBLE</td>
<td>33</td>
<td>304</td>
<td>1.00</td>
<td>5</td>
</tr>
<tr>
<td>DAYS</td>
<td>419</td>
<td>2624</td>
<td>.95</td>
<td>83</td>
</tr>
<tr>
<td>FIND</td>
<td>58</td>
<td>953</td>
<td>.99</td>
<td>32</td>
</tr>
<tr>
<td>GCD</td>
<td>325</td>
<td>4747</td>
<td>.99</td>
<td>58</td>
</tr>
<tr>
<td>TRITYP</td>
<td>420</td>
<td>862</td>
<td>.99</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 22 shows the results of this "multiple satisfaction" method of test case reduction. The first three columns repeat the previous results of generating test cases without reduction. The next column gives the number of test cases generated by multiple satisfaction. The last two columns give the number of mutants the reduced set of test cases killed and the resulting mutation score.

The decrease in mutation score by the reduced set was not large (between one and seven percent). This means that it is possible to trade off a large amount of test case creation for a small amount of testing strength.
To see how much improvement is achieved, we must compare the number of mutants killed as well as the number of test cases created. Let the number of test cases generated with no constraint combination be $T$ and the number generated with constraint combination be $T'$. The number of mutants killed by $T$ is $K$ and the number killed by $T'$ is $K'$. The ratios of these numbers can be defined as:

$$\sigma = \frac{T'}{T}, \quad \nu = \frac{K'}{K}$$

Our goal when reducing the number of constraints is to reduce the ratio of test cases generated ($\sigma$) without reducing the ratio of mutants killed ($\nu$). In a (hypothetical) perfect case, we would reduce the number of test cases to one without effecting the number of mutants killed ($\sigma = 1/T$ and $\nu = 1$). In the worst case, we would reduce the number of mutants killed to zero without lowering the number of test cases ($\sigma = 1$ and $\nu = 0$). We can define the reduction coefficient ($R$) as:

$$R = \frac{\nu}{\sigma} = \frac{K'}{K} \cdot \frac{T}{T'}$$

$R$ grows larger as the number of test cases decreases and grows smaller as the number of mutants killed decreases. As the number of test cases approaches one, $R$ approaches infinity (perfection) and as the number of mutants killed approaches zero $R$ approaches zero. Table 23 shows values for $R$ calculated for this experiment.

<table>
<thead>
<tr>
<th>Program</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE</td>
<td>6.4</td>
</tr>
<tr>
<td>DAYS</td>
<td>5.0</td>
</tr>
<tr>
<td>FIND</td>
<td>1.8</td>
</tr>
<tr>
<td>GCD</td>
<td>5.5</td>
</tr>
<tr>
<td>TRITYP</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Table 23. Reduction Coefficient of Combining Constraints Experiments

The reduction coefficient was well above 1 for each of the five programs. If $R = 1$, then there would have been no improvement, so that can be regarded as a cutoff point. The fact that $R$ varies so much indicates that we can expect very different amounts of improvement from combining constraints depending on the program being tested. Note that the number of mutants killed was not changed appreciably for any of the five programs—the variation was mainly due to the number of test cases created. TRITYP, whose reduction coefficient was above 10, had over 90% fewer test cases created, whereas FIND only had about 45% fewer test cases.

**Improve Satisfaction**

Another area of inefficiency of this testing technique is in the satisfaction of the constraints. Chapter V discussed several algorithms for satisfying constraints. As mentioned, most satisfaction techniques use some sort of searching strategy to find a solution. Table 24 gives the number of "searches" that the domain reduction algorithm (Algorithm 6) used in generating the test data for each of the five programs. Each search represents an attempted solution to a set of constraints that was found and either used or discarded.
Table 24. Number of Satisfaction Attempts

<table>
<thead>
<tr>
<th>TCs</th>
<th>Attempts</th>
<th>Attempts per TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE</td>
<td>32</td>
<td>352</td>
</tr>
<tr>
<td>DAYS</td>
<td>419</td>
<td>756</td>
</tr>
<tr>
<td>FIND</td>
<td>58</td>
<td>7960</td>
</tr>
<tr>
<td>GCD</td>
<td>325</td>
<td>2783</td>
</tr>
<tr>
<td>TRITYP</td>
<td>420</td>
<td>16251</td>
</tr>
</tbody>
</table>

The second column gives the number of attempts per test case created. Three of the attempts per test case are less than 15, which seems reasonable. The number of attempts per test case for both TRITYP and FIND are higher than this. TRITYP has a comparatively high branching factor, so the path expressions were more complicated than the other programs.

FIND seems an anomaly because of the much higher number of search attempts that Godzilla made. It is not clear why this number is so high. There were many more failures when satisfying the constraints for FIND, so the search limit was reached more often. Other than that, it seems that the constraints must somehow be harder to satisfy for FIND than for the other programs.

To try to answer why the constraints for TRITYP and FIND were so much harder to satisfy, the 3600 constraints generated for the five programs were analyzed. It seems likely that the difficulty of satisfying the constraints is related to some "complexity" of the constraints. There are not many data points (five programs), and there are many factors that contribute to a constraint complexity (number of conjuncts, disjuncts, variables per conjuncts, constraints per conjuncts, etc.), so it is not surprising that there is no clear, obvious correlation between these factors and the number of satisfaction attempts. Table 25 shows two factors that are intuitively likely to be related.

Table 25. Satisfaction Attempts Versus Constraints

<table>
<thead>
<tr>
<th></th>
<th>Attempts per TC</th>
<th>Constants per Constraint</th>
<th>Variables per Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND</td>
<td>137.2</td>
<td>0.4</td>
<td>8.3</td>
</tr>
<tr>
<td>TRITYP</td>
<td>38.7</td>
<td>8.7</td>
<td>24.8</td>
</tr>
<tr>
<td>BUBBLE</td>
<td>11.0</td>
<td>3.9</td>
<td>3.6</td>
</tr>
<tr>
<td>GCD</td>
<td>8.6</td>
<td>11.7</td>
<td>20.0</td>
</tr>
<tr>
<td>DAYS</td>
<td>3.3</td>
<td>0.8</td>
<td>2.9</td>
</tr>
</tbody>
</table>

The number of constants per test case for FIND is the lowest of all five programs, which may contribute to the difficulty of satisfying the constraints for FIND. This is intuitively pleasing, because the discussions of satisfaction algorithms in Chapter V indicated that constant values should make constraints easier to satisfy. Unfortunately, this value is also low for DAYS, which had the "easiest" constraints to satisfy, indicating that the number of constants per test case is not the only factor contributing to the difficulty of satisfying the constraints. On the other hand, the number of variables per constraint was the lowest for DAYS. This fits our intuition that constraints with a large number of variables should have more interaction and require more searching for a solution.

These two factors seem related to the difficulty of satisfying constraints, but it is not obvious that they are entirely the determining factors. Probably a complete measure of constraint complexity would depend on not only these, but other factors, including the number of conjuncts in the constraint system, and other measures. So at this point it seems clear that a satisfaction algorithm that makes fewer searching attempts would improve the overall efficiency of the constraint based technique, but we cannot determine before satisfaction how difficult a constraint system will be to satisfy.
CHAPTER X

CONCLUSIONS AND SUGGESTIONS FOR FURTHER STUDIES

No one who works a forty hour week is ever going to beat me.

— Bill Rogers

In this dissertation, a new testing technique has been presented. It is an error based method for automatically generating mutation-adequate test data. The technique is heavily based on mutation analysis and solves a major problem with using mutation analysis to test software—that of generating the test data. Because this is a new solution to a very general problem, this technique has several aspects and applications that have not been explored in this dissertation. This chapter presents five areas of possible future research:

1. Finding the most effective mode of operation for the test case generator.
2. Finding an improved algorithm or set of algorithms for satisfying the constraints.
3. Using the program constraints to test software specifications.
4. Integrating the constraint-based technique with software development environments.
5. Extending the constraints to allow more detection of equivalent mutants.

These issues are discussed in the following paragraphs.

Most Effective Modes of Operation

During most of the research for this dissertation, very little attempt was made to optimize the way test cases were generated. Several of the experiments in Chapter IX indicated that it would be relatively simple to reduce the number of test cases generated—thus reducing the execution time. To be worthwhile, the time spent reducing the number of test cases must be less than the time saved, and the overall effectiveness of the test cases must not be significantly reduced. Of course, if significant time savings could be made with only slight loss in effectiveness, the testing power lost could well be worth it.

The most obvious way to reduce the number of test cases would be to perform some random sampling from the set of test cases generated during execution. Probabilistic sampling of mutant operators has been found to be effective in reducing the number of mutants generated during mutation analysis and can be expected to be similarly effective in reducing the number of test cases generated [Acre80]. Sampling from the test case set takes advantage of the redundancy of the test cases, but in a very simple-minded way.

The experiments described in Chapter 9 indicated that there is some hope of reducing the number of test cases by selecting test cases created by certain mutation operators. This takes advantage of another form of redundancy in the mutant operators. Though the results look positive, it is not clear whether the redundancy always exists, so this approach could be error-prone. An interesting question asks whether we can characterize when this redundancy does exist.

Combining constraints is a way of directly reducing the amount of redundancy in the test cases. From the experiment described in Chapter IX, combining constraints can be a powerful method of reducing the number of test cases in a way that by definition will not significantly lower the adequacy of the test cases. In Budd’s dissertation [Budd80a], he suggested the possibility of algorithmically finding
redundancy in the mutations generated by a mutation system. Since the constraints in some sense capture the intent of each mutation, constraint duplicity may well be the algorithmic redundancy finder he hoped for. This has the potential for being a very expensive approach, as sophisticated algebraic algorithms tend to be slow. The expense may be worthwhile though, since the test cases are generated only once and then executed many times.

Satisfaction of Constraints

It seems apparent that there is no perfect algorithm for solving arbitrary mathematical constraints. Several choices were presented in Chapter V, all with their individual advantages and disadvantages. The method used to satisfy the constraints in the Godzilla implementation is adequate but does have some shortcomings. Specifically, it does not detect all instances of infeasible constraints, it does a poor job of recognizing when constraints are too complicated for it to solve, and it converges slowly if at all when constraints are very complex. This last problem makes one wonder if this algorithm will be practical with larger scale software.

When constraints have a simple form (linear, few clauses, etc.) then a simple satisfaction algorithm is probably the best approach. These algorithms do not have the overhead that may be involved with more complex algorithms. One solution is to offer a range of satisfiers and use the appropriate algorithm for the appropriate sets of constraints. The important question is whether or not we can decide prior to satisfaction which algorithm would be best to use for the constraints given, or whether we would have to start with the simpler (and faster) techniques and move to progressively more complicated satisfiers as the simpler ones fail. Obviously the latter is a more expensive strategy.

One other suggestion has been to involve the tester in the satisfaction process. Rather than giving up when the constraints are difficult to solve, the satisfier could ask the tester for help. Often humans can see patterns and provide simple solutions to global patterns whereas a computer program can provide the detail solutions. In particular, the breaking of cycles (as discussed in Chapter V), is usually very easy when done intuitively despite being quite difficult to do algorithmically. The technical difficulty with using the tester to help solve the constraints is presenting the constraints in terms that are easy for a human to understand and (conceptually) manipulate. Since constraints usually model something that we are familiar with (such as test cases), it is often easier to view constraints in geometric forms or even some other, higher level, representation of the constraints.

Testing Program Specifications

One problem that seems amenable to being solved by constraints is that of testing software specifications. A difficulty with writing specifications for software has been that even though the process can contribute to the actual software being correct with respect to the specifications, there is nothing that assures us that the specifications are correct. The idea of using constraints to test specifications is quite simple, but does require that the software has already been written.

The process is simple and goes as follows. Given the program, compute the path expression constraint \( PE \) for a statement \( S \) and the necessity constraint \( C \) for mutant \( M \). Combine these to get \( PE \land C \). Then satisfy \( PE \land C \) to produce a test case \( t_{PE} \) that kills \( M \). This is the test data generation strategy that has been described in this dissertation.

The next step is to use the specifications, which are assumed to be given in the form of assertions on the program. Replace \( PE \) with the assertion \( A \) given by the specifications to get \( A \land C \). Generate a test case \( t_{A} \) to satisfy \( A \land C \). Next, execute \( t_{A} \) on the original program and verify that the output is correct. The final step is to execute \( t_{A} \) on the mutant \( M \) that died from \( t_{PE} \).

If \( M \) is not killed by \( t_{A} \), then the assertions generated from the specifications do not agree with the constraints generated from the program—and the program does not agree with the specifications. If this happens, then one of three things is true. Either the constraint is wrong, the program is wrong, or the specification is wrong. We assume the first is under our control and can at least be easily checked.
we already verified that the output of the program on both $i_A$ and $i_{PE}$ was correct, then the specification must be wrong.

This is most easily demonstrated with an example. In Figure 28 is a version of the MAX function with an $avr$ mutation. This version of MAX accepts an array of integers and returns the largest element of the array. To the right of the mutant program are the path expression, the specification assertion, and the necessity constraint.

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Mutant Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION MAX (A)</td>
<td>FUNCTION MAX (A)</td>
</tr>
<tr>
<td>INTEGER A(5)</td>
<td>INTEGER A(5)</td>
</tr>
<tr>
<td>MAX = 1</td>
<td>MAX = 1</td>
</tr>
<tr>
<td>DO 10 I = 2,5</td>
<td>DO 10 I = 2,5</td>
</tr>
<tr>
<td>IF (A(MAX).LT.A(I))</td>
<td>IF (A(MAX).LT.A(I))</td>
</tr>
<tr>
<td>MAX = 1</td>
<td>MAX = MAX</td>
</tr>
<tr>
<td>10 CONTINUE</td>
<td>10 CONTINUE</td>
</tr>
</tbody>
</table>

\[ PE: A_i > A_{MAX} \land I > 1 \]
\[ A: A_i < A_{MAX} \land I > 1 \]
\[ C: MAX \neq I \]

Figure 28. Array Version of Max—Specification Example

Figure 29 shows test cases produced from both the path expression constraint and the specification constraint. As can be seen, the test case $i_{PE}$ is adequate to kill the mutant, whereas $i_A$ is not. Thus the faulty specification is discovered.

<table>
<thead>
<tr>
<th>Expression</th>
<th>$PE \land C$</th>
<th>$A \land C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint</td>
<td>$A_i &gt; A_{MAX} \land I &gt; 1$</td>
<td>$A_i &lt; A_{MAX} \land I &gt; 1$</td>
</tr>
<tr>
<td>Test Case</td>
<td>$i_{PE}=(A=1 2 3 4 5)$</td>
<td>$i_A = (A=5 4 3 2 1)$</td>
</tr>
<tr>
<td>Program Result</td>
<td>MAX=5</td>
<td>MAX=5</td>
</tr>
<tr>
<td>Mutant Result</td>
<td>MAX=1</td>
<td>MAX=5</td>
</tr>
</tbody>
</table>

Figure 29. Results of Specification Example

Obviously there are a lot of questions still to be answered about testing specifications with constraints. At some level, what we are really doing is checking to see if the program matches the specifications. Although it seems that it would be more useful if we could test the specifications before producing the software, this way of testing specifications could possibly prove beneficial. In particular, the fact that this method is highly automatable is very attractive.

Join with Software Development Environments

Various researchers are currently building programming environments that partially automate or at least make easier the process of writing software [Reps82, Reps84]. Much of this work has been focused on trying to collapse the so-called "Edit-Compile-Debug" cycle so that compilation is performed automatically when the source code is changed. The most common way of doing this is by incrementally maintaining an up-to-date version of some internal form of the code.

A simple extension of this effort would be to include testing in the collapsible cycle—as in the "Edit-Compile-Test-Debug" cycle. Constraints offer an easy method for extending the consistency
maintenance of object code to consistency maintenance of the test cases. As the source is changed, the constraints can be modified to reflect this change and test cases can be updated. The new test cases could then be automatically executed and test results reported to the programmer in whatever way is appropriate.

**Extending Equivalent Mutant Detection**

In Chapter VII, various methods for detecting equivalent mutants were surveyed and a new method, based on constraints, was proposed. It is not obvious that this new method subsumes all of the other equivalent detection methods, but it does seem possible and perhaps likely. This partly depends on improved satisfaction techniques that are better at recognizing infeasible constraints. Equivalent mutant detection can also be aided by stronger path expressions. For example, the Algorithm I from Chapter IV can be extended to a two pass algorithm to increase the number of paths computed. Equivalent mutant detection also can be aided by the addition of constraints specifically designed to make infeasible constraints more easily recognizable. These constraints need not have any value for generating test cases other than making infeasible constraints recognizable.

**Summary**

Although mutation analysis has been repeatedly demonstrated to be an effective method for testing software [DeMi86, Girg86, DeMi89], a shortcoming has always been that it does not generate test data. By generating test data specifically to kill mutant programs, we get test data with the same quality as data generated interactively but with none of the pain and expense of that interaction. Of course, without a solution to the problem of sufficiency, the test data will not be fully adequate. This problem is certainly not unique to this method or to mutation analysis. As a matter of fact, Richardson and Thompson [Rich88] are working on just this problem: there is no reason why a solution could not be integrated with a tool that generates test data using constraints.

Because this technique is based on a set of rules, these rules can be modified to reflect new research in software testing, knowledge about the relevance of the rules, new languages, or a host of other innovations. In particular, the capabilities of other testing techniques can be incorporated simply by expanding the set of necessity constraint templates. Therefore, it is the author's firm belief that this technique is currently the most effective method known for testing software, at least at the procedural level. Whether this technique can be applied to large integrated programs is still an open question, as indeed is the larger question of whether we even wish to test integrated software by generating test data.

Because this technique is based on the construction and solution of algebraic constraints that describe the test cases, it has been called the constraint-based testing technique. These constraints are formulated from the test program and are in terms of program names and operators. The constraints represent ranges of values and relationships among variables that are necessary to demonstrate particular errors.

To demonstrate this technique, a prototype test data generator has been implemented that generates the constraints and solves them to generate test case values. This generator, called Godzilla, has been integrated with the Mothra testing system and used to demonstrate the usefulness of the technique and to experiment with various ways of using the technique.

This dissertation exemplifies a particular philosophy towards solving problems in software engineering. Many of the problems we face in software engineering tend to be theoretically very difficult—NP-complete or worse. This means that complete solutions are often not feasible. On the other hand, great benefits can be gained by producing partial solutions; either that only work in certain situations, or that occasionally fail, or that produce results that are less than optimal. These partial solutions to problems are important to software engineering and represent much of the work that has been done in the field [DeMi79a]. Providing partial solutions to problems is a major theme in this dissertation, both in terms of the general problem of software testing and in terms of the individual technical problems that were solved to make the constraint-based technique a reality. Unfortunately, partial solutions to
problems do not lend themselves to theoretical analysis, so we are left with experimentation as the primary method for verifying our results. This is a second major theme of this dissertation, that experimental results must be used to verify that the technique is sound. Many others have used experimental work to demonstrate ideas in software testing [Burn78, Budd80, DeMi80a]. Naturally, experimentation requires a laboratory and the Mothra and Godzilla systems have served as the laboratory for most of the research.

This work presents a new and very powerful technique for generating test data. The technique is, in the author's knowledge, the first attempt to generate data based on adequacy. It solves a major problem with using mutation analysis as a practical method for testing software—that of creating test data. The method for generating test data has been fully implemented and integrated with Version 1.2 of the Mothra testing system—a mutation analysis based testing system for Fortran 77. The Godzilla implementation is composed of over 15,000 statements in the C programming language that includes the ability to create necessity constraints to kill mutants, perform symbolic executions of Fortran 77 programs to derive reachability constraints, and satisfy the constraints to generate test cases for the test program.

Because of the mutation analysis basis, the test data generated includes the error detection capabilities of such test case generation methods as branch analysis and domain analysis. The experiments that are described in this paper verify that the test data generation technique creates high quality test cases that score well on the mutation system. The experiments in Chapter IX show that the technique is at least competitive with other test data techniques and will be more powerful. Moreover, because the necessary constraints are derived from rules to describe test cases, the technique can be easily extended to handle other types of faults. These experiments also show that this method is a much more cost effective means of creating mutation-adequate test sets than by the current human methods.
BIBLIOGRAPHY


[Budd77] Budd, T. and F. Sayward. Users Guide to the Pilot Mutation System. TECHNICAL REPORT #114, Department of Computer Science, Yale University, 1977.

[Budd80] Budd, T., R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. TECHNICAL REPORT GIT-ICS-80/01, Georgia Institute of Technology, 1980.


[Chuo87] Chuah, T. Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing. IEEE TRANSACTIONS OF SOFTWARE ENGINEERING 13, no. 5 (May 1987).


[DeMil87a] DeMillo, R. A., W. M. McCracken, R. J. Martin, and J. F. Passafiume. Software


Howden, W. E. Reliability of the Path Analysis Testing Strategy. TRANSACTIONS OF SOFTWARE ENGINEERING (IEEE) 2, no. 3 (September 1976).


Mutations. RESEARCH REPORT #276, Department of Computer Science, Yale University, 1979.


[Clar76] Clarke, L. A. A System to Generate Test Data and Symbolically Execute Programs. TRANSACTIONS ON SOFTWARE ENGINEERING 2, no. 3 (September 1976): 215-222.