Automatically Detecting Equivalent Mutants and Infeasible Paths [*]

A. Jefferson Offutt
ISSE Department
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
fax: 703-993-1638
email: ofut@isse.gmu.edu

Jie Pan
Mail Stop C204
PRC, Inc.
12005 Sunrise Valley Drive
Reston, VA 22091
phone: 703-620-8604
email: jpan@isse.gmu.edu

October 6, 1998

Abstract

Mutation testing is a technique for testing software units that has great potential for improving the quality of testing, and thereby increasing our ability to assure the high reliability of critical software. It will be shown that recent advances in mutation research have brought a practical mutation testing system closer to reality. One recent advance is a partial solution to the problem of automatically detecting equivalent mutant programs. Equivalent mutants are currently detected by hand, which makes it very expensive and time-consuming. The problem of detecting equivalent mutants is a specific instance of a more general problem, commonly called the *feasible path problem*, which says that for certain structural testing criteria some of the test requirements are infeasible in the sense that the semantics of the program imply that no test case satisfies the test requirements. Equivalent mutants, unreachable statements in path testing techniques, and infeasible DU-pairs in data flow testing are all instances of the feasible path problem. This paper presents a technique that uses mathematical constraints, originally developed for test data generation, to automatically detect some equivalent mutants and infeasible paths.

Keywords—software testing, mutation testing, constraints, feasible path analysis

The Journal of Software Testing, Verification, and Reliability, Vol 7, No. 3, pages 165–192, September 1997.

1 Introduction

Mutation testing is a technique, originally proposed by DeMillo et al. [DLS78] and Hamlet [Ham77], that requires testers to create test data that cause a finite, well-specified set of faults to result in failure. The testers do this by finding test cases that cause faulty versions of the program to fail. These test cases will then either result in correct output from the test program (demonstrating the

[*]Partially supported by the National Science Foundation under grant CCR-93-11967.

absence of those types of faults) or cause the test program to fail (detecting a fault). The technique thus serves two goals: it provides a test adequacy criterion, and leads to detection of faults in the program being tested.

Unit level testing techniques such as mutation hold great promise for improving the quality of software. Although most of these techniques are currently so expensive that practical use is rare, recent advances in mutation research has brought practical mutation testing system closer to reality. Commercial tools (e.g., PiSCES [VMM91]) have recently become available, although they are still not widely used in industry. A major problem with mutation testing is that it is too expensive to apply; one aspect of the cost is identifying programs that are intended to be faulty, but in fact are not. These programs are said to be *equivalent*. Previous papers have presented ways to speed up mutation testing [How82, OLR+96, DO91, UOH93], including a technique for partially solving the problem of automatically detecting equivalent mutants that was based on compiler optimization techniques [OC94]. Equivalent mutants are currently detected by hand, which makes it very expensive and time-consuming. This paper presents a new technique for partial detection of equivalent mutants based on constraint-based testing that gives much better results than the compiler optimization techniques.

The rest of this paper presents a method to detect equivalent mutants. The remainder of the introduction provides background material on mutation testing, presents the problem of recognizing equivalent mutants, and discusses previous work on this problem. Section 2 describes constraint-based testing, then provides full details, including algorithms, on how to use constraints to detect infeasible constraint systems, and therefore equivalent mutants. Next, a proof-of-concept tool that implements the equivalent mutant detection strategies is discussed, then empirical results from using the tool are presented, and conclusions and several specific suggestions for improving the technique and the tool are given.

1.1 Mutation Testing Overview

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

Figure 1 shows a short function and five mutations; the original program is shown to the left, and the mutant programs are represented on the right by the lines preceded by the Δ symbol. Note that each mutated statement represents the change that is made to create one mutant program. A mutation *operator* is a rule that is applied to a program to create mutants. The Mothra mutation system [DGK+88] uses 22 mutation operators to test Fortran 77 programs, as shown in Table 1. The Mothra operators replace each operand by all other syntactically legal operands ($\Delta 1$, $\Delta 3$ and $\Delta 5$ in Figure 1), modify expressions by replacing operators and inserting new operators ($\Delta 2$), and modify entire statements ($\Delta 4$).

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

```
      Original Program                       With Embedded Mutants

FUNCTION Min (I, J : Integer)         FUNCTION Min (I, J : Integer)
         RETURN Integer IS                     RETURN Integer IS
   MinVal :  Integer;                    MinVal :  Integer;
BEGIN                                 BEGIN
   MinVal := I;                          MinVal := I;
   IF (J < I) THEN                Δ1     MinVal := J;
       MinVal := J;                      IF (J < I) THEN
   END IF;                        Δ2     IF (J > I) THEN
RETURN (MinVal);                  Δ3     IF (J < MinVal) THEN
End Min;                                     MinVal := J;
                                  Δ4         TRAP;
                                  Δ5         MinVal := I;
                                         END IF;
                                      RETURN (MinVal);
                                      End Min;
```

Figure 1: Function MIN

the test case is kept as an "effective" test case.

In Figure 1, note that the mutant $\Delta 3$ replaces the variable `I` with `MinVal`. In the previous statement, the value of `I` was assigned to `MinVal`. At this point in the program, `I` and `MinVal` will always have the same value, thus $\Delta 3$ represents an equivalent mutant.

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

1.2 Equivalent Mutant Problem

This paper focuses on the problem of detecting equivalent mutants, which has been an obstacle to the practical application of mutation. Without detecting all the equivalent mutants, the mutation score will never be 100%. Thus, the tester will not have complete confidence in the program and the test data. Worse, the tester will be left wondering whether the remaining mutants are equivalent or if the test set is insufficient. Detecting equivalent mutants by hand is very time-consuming, which contributes to making the cost of mutation testing prohibitively high.

1.2.1 Distribution of Equivalent Mutants Among Mutant Types

Equivalent mutants are not evenly distributed among the mutant types; they tend to cluster among only a few types. Table 2 summarizes statistics from the programs used in section 3.2 of this paper. The first column in the table gives the mutant operator type and the second column gives the

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

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

percentage of the total number of equivalent mutants represented by each type. The third column gives the percentage of all mutants that are equivalent of that type – in the programs studied, 9.1% of the mutants were equivalent.

Offutt and Craft [OC94] and Budd [Bud80] give similar statistics of the distribution of equivalent mutants among mutant types. Both sources indicate one very interesting fact. One mutant type, *absolute value insertion* (*abs*), has many more equivalent mutants than any other mutant type. The *abs* mutant operator inserts three unary operators before each expression — ABS computes the absolute value of the expression, NEGABS computes the negative of the absolute value, and ZPUSH kills the mutant if the expression is zero, otherwise does nothing (this forces the tester to cause each expression to have the value zero, a common testing heuristic). The fact that equivalent mutants are clustered among the *abs* mutants is used in the techniques presented in this paper.

1.2.2 Do Procedures for Automatically Detecting Equivalence Exist?

Budd and Angluin [BA82] examine the relationships between equivalence and test data generation. They prove that if there is a computable procedure for checking if two programs are equivalent, there is also a computable procedure for generating adequate test data for a program and vice versa.

Mutant Type	% of Equivalent	% of All Mutants
ABS	47.19	4.30
ACR	14.10	1.28
SCR	7.05	0.64
UOI	6.04	0.55
SRC	4.89	0.45
SVR	4.46	0.41
ROR	3.60	0.33
SDL	2.16	0.20
CRP	1.58	0.14
AAR	1.44	0.13
RSR	1.44	0.13
LCR	1.15	0.10
ASR	1.15	0.10
CSR	1.01	0.09
SAR	1.01	0.09
All others	1.73	0.16
Total	100.00	9.10

Table 2: **Equivalent Mutants among Mutant Types**
Programs Used in This Paper

They also show that, in general, neither of these computable procedures exists. Thus, there cannot be a complete algorithmic solution to the equivalence problem. That is, detecting equivalence either between two arbitrary programs or two mutants is an undecidable problem.

Fortunately, the equivalent mutant problem has one advantage over the general equivalence problem. Specifically, it is not necessary to determine the equivalence of arbitrary pairs of programs. Because of the definitions of mutation operators, mutants are syntactically very much like their original program. Although Budd and Angluin also prove that this problem is undecidable, it has been suggested that for many specific cases, equivalence can be decided [Acr80, OC94]. This paper reports results from techniques and heuristics that automatically detect many of the equivalent mutants.

1.2.3 Previous Work in Detecting Equivalent Mutants

It is obvious that automatically detecting equivalent mutants can save much time and energy for the testers, but Acree [Acr80] found that it could also prevent people from making errors in marking equivalent mutants. In a study of 50 mutants, half of which were equivalent, Acree found that people judged mutant equivalence correctly only about 80% of the time. The people marked equivalent mutants non-equivalent (type 2 errors) 12% of the time and non-equivalent mutants equivalent (type 1 errors) 8% of the time. Because type 2 errors can be corrected during later testing a conservative approach would be to try to eliminate type 1 errors, even if that means allowing a few type 2 errors. If an automated tool is used, the effect of the types of errors depends on how the tool is used. Type 1 errors result in the mutation score being overestimated, and type 2 errors result in the mutation score being underestimated. If all mutants that are **not** determined to be equivalent are examined by hand, then the remaining equivalent mutants can be determined by hand. If the remaining mutants are not examined by hand (as suggested in this paper) then type 1 errors will result in an overestimate of the number of equivalent mutants, hence an overestimate of the mutation score. Likewise, type 2 errors will result in an underestimate of the mutation score. Thus, type 1 errors could be viewed as being less conservative. The advantage of using automated techniques to detect equivalent mutants is that they can be more conservative by avoiding as many type 1 errors as possible.

Baldwin and Sayward [BS79] proposed using compiler optimization techniques to detect equivalent mutants. The key intuition behind this approach is that code optimization transformations produce equivalent programs, and some equivalent mutants are, in some sense, either optimizations or de-optimizations of the original program. So when a mutant satisfies a code optimization rule, algorithms can detect that it is equivalent. Baldwin and Sayward describe six types of compiler optimization techniques that can be used to detect equivalent mutants.

Offutt and Craft [OC94] designed algorithms for these six techniques, and developed and implemented Equalizer, an automated tool for detecting equivalent mutants. They found that Equalizer detected an average of 10% of the equivalent mutants for 15 programs.

In his PhD dissertation [Off88], Offutt presents a technique for using mathematical constraints for testing, which is called Constraint-Based Testing (CBT). How constraints can be used to generate test cases to satisfy mutation testing was presented in a paper with DeMillo [DO91]. The dissertation also suggested using CBT to detect equivalent mutants, but did not offer details for how to do it. This paper develops the idea, presents specific strategies and algorithms for detecting equivalent mutants, and presents results from an implementation of these algorithms.

1.3 Feasible Path Problem

For most structural testing criteria, some of the test requirements are infeasible in the sense that the semantics of the program imply that no test case exists that meet the test requirements. Goldberg, Wang, and Zimmerman [GWZ94] define the problem as follows: "given a description of a set of control flow paths through a procedure, *feasible path analysis (FPA)* determines if there is input data that causes execution to flow down some path in the collection". We generalize this to the *feasible test problem (FTP)*: given a requirement for a test case, the feasible test problem is to determine if there is input data that can satisfy the requirement. Feasible path analysis is one instance of this problem. Mutation provides another – the FTP is to decide whether the mutant is equivalent or killable.

The problem also arises in statement coverage, where the FTP is whether the statement is reachable, branch coverage techniques [Mye79], where the FTP is whether a branch, predicate, condition, or combination of conditions can be covered, and in data flow testing [FW88], where the FTP is whether a definition-clear subpath can be found between a DU-pair. Unfortunately, this problem is formally undecidable [GWZ94, DO91] for all variations mentioned here, thus approximations must be used.

The most recent work on the feasible test problem is by Goldberg et al. [GWZ94] and Jasper et al. [JBW$^+$94]. Goldberg et al. presents a structural testing system that uses a theorem prover to attempt to decide if test requirements can be satisfied. Jasper et al. also use theorem proving techniques, and attempt to determine the feasibility of expressions containing references to arrays, and apply their techniques to the modified condition decision coverage criterion.

This paper focuses on the FTP within the context of mutation testing, and uses a very different approach to determine feasibility. Specifically, a heuristic-based set of transformations is applied to mathematical constraints to look for infeasibility. These transformations, and their application to infeasibility, are formally proved in the following sections.

2 Using Constraints to Detect Equivalent Mutants

In this paper, a *constraint* is a mathematical algebraic expression that restricts the input space of the program to be the portion of the input domain that satisfies a certain goal. As a simple example, $(x > 0)$ describes the portion of the input domain where x is positive. More complicated constraints can be used to describe higher-level goals, such as an array must be sorted or a shape represented by a set of points must be rectangular.

Constraint-based testing (CBT) [DO91] uses constraints for automatic test data generation. In CBT, a constraint represents the conditions under which a mutant will die. The technique in this paper uses the fact that if a test case kills the mutant, the constraint system will be satisfied by that test case. If the constraint system cannot be true, then there is no test case that can kill the mutant and the mutant is equivalent. The general approach to using constraints to detect equivalent mutants is to look for infeasibility in constraint systems.

This section gives a detailed discussion of how constraints can be used to detect equivalent mutants, and how the procedure works. Since constraint-based testing was previously used for test data generation, this technique is introduced with constraint-based test data generation. Then proofs are given for three theorems that show how CBT can be used to detect equivalent mutants.

2.1 The Constraint-based Testing Technique

Constraint-based testing (CBT) uses the concepts of mutation analysis to automatically create test data. This test data is designed specifically to kill the mutants of the test program. Such test data can be used to kill mutants within a mutation system. For each mutant, a test case is said to be *effective* if it causes the mutant to fail, and a test case is *ineffective* if it does not. For the following, P is a program, M is a mutant of P on statement S, and t is a test case for P. The *state* of a program is as usual; the values of all data items and the program counter. To kill M, t has to have three broad characteristics:

- **Reachability:** Since a mutant is represented as a syntactic change to an executable statement, and the other statements in the mutated program are syntactically equal to the statements in the original program, a minimum requirement for a test case to kill the mutant is that it execute the mutated statement. This characteristic is called the *reachability condition.*

  If t cannot reach S, it is guaranteed that t will not kill M [DO91].

- **Necessity:** For a test case to kill a mutant, it must be able to cause the mutant to have an incorrect state if it reaches the mutated statement. This characteristic is called the *necessity condition.*

  For t to kill M, it is *necessary* that if S is reached, the state of M immediately following some execution of S must be different from the state of P at the same point [DO91].

  To see why, note that M is syntactically equal to P except for the mutated statement S. Thus, if the states of the two programs do not differ after some execution of S, they will never differ. If S is in a loop, necessity requires that the state be incorrect after any given execution of the mutated statement. Of course, the incorrect state must propagate through the program to the last execution of the statement, but that requirement is part of sufficiency.

- **Sufficiency:** The *sufficiency condition* is that the final state of M differs from that of P [DO91].

Note that the necessity condition specifically does not include the reachability condition. That is, a test case can satisfy the necessity condition even if it does not actually execute the statement. Although this may seem counter-intuitive, the orthogonality makes the necessity constraints easier to create by automated tools.

Let D represent the entire domain of all test cases t for P. In light of the three conditions above, D can be divided in several ways for each mutant:

- $D = D_r \cup D_{\overline{r}}$, where D_r is the portion of D that will satisfy the *reachability condition* C_r for a given mutant and $D_{\overline{r}}$ is the portion of D that will not.
- $D = D_n \cup D_{\overline{n}}$, where D_n is the portion of D that will satisfy the *necessity condition* C_n for a given mutant and $D_{\overline{n}}$ is the portion of D that will not.
- $D = D_s \cup D_{\overline{s}}$, where D_s is the portion of D that will satisfy the *sufficiency condition* C_s for a given mutant and $D_{\overline{s}}$ is the portion of D that will not.

The letter C stands for a condition. The subscripts r, n, and s are taken from the terms reachability, necessity and sufficiency. ¿From the definitions of the above three conditions and sub domains of P, the following facts are observed:

Fact 1: t is an effective test case that will kill M if and only if $t \in D_s$ for M,

Fact 2: if t is an effective test case that will kill M then $t \in D_r \cap D_n$,

Fact 3: $D_s \subseteq D_r \cap D_n$.

A test case from the intersection of D_r and D_n will reach the mutated statement and cause an incorrect intermediate state to be created. But this incorrect state may not always propagate through the execution to result in incorrect output. Morell [Mor90] discusses various reasons for this, including masking of data values and error correction. Thus the set of test cases that cause incorrect output is actually a subset of $D_r \cap D_n$.

Unfortunately, finding t such that $t \in D_r$ is an undecidable problem. This is because determining whether t executes S is reducible to the halting problem. Thus, a weaker condition is defined. C_R is defined such that if S is executed, then C_R is true. D_R is the domain that contains all inputs t that satisfy C_R. Of course, a trivial solution to C_R is $C_R = true$, but in practice we strive to find the strongest reachability condition possible. Since $C_r \Rightarrow C_R$, it is clear that:

Fact 4: $D_r \subseteq D_R$.

Figure 2 is a Venn diagram that graphically shows these domains and their relationships for one mutant. The tests that kill the mutant (D_s) are located in the intersection of the tests that reach the mutant (D_R) and the tests that satisfy necessity (D_n). The above facts and this diagram are used in later proofs.

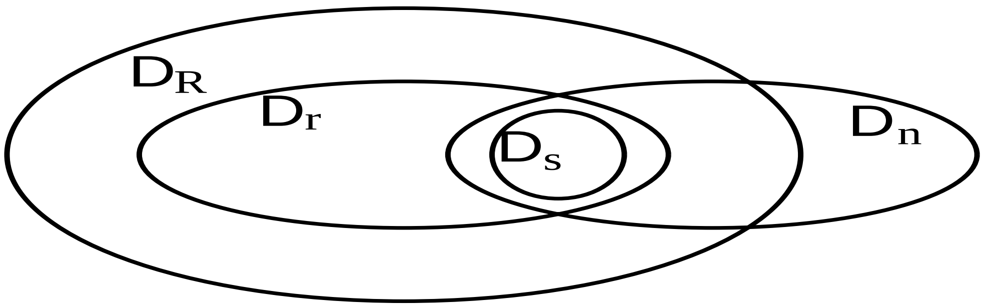

Figure 2: **Input Domain Subsets**

CBT uses a *path expression* to describe the reachability condition (the weaker condition), C_R, for a statement. A path expression for a statement S in a program P is an algebraic expression that describes a condition on test cases that will be true when P reaches S. Path expressions usually describe multiple paths to S by using a disjunctive formula, where each clause represents a separate path. Path expressions are automatically derived from the program by extracting the predicate expressions on the program's control flow graph. There are a variety of ways to do this, and a variety of analysis techniques, most of which are based on symbolic evaluation, are used to refine or simplify the expressions. As should be expected, more effort in the analysis usually leads to more successful results.

Since mutation operators represent syntactic changes, a test case that satisfies the necessity condition must ensure that the syntactic change effected by the mutation results in an incorrect state for the program. CBT uses a *necessity constraint* to describe this necessity condition C_n. That is, if a test case that satisfies the necessity condition will cause the mutated statement to be reached, the state immediately following some execution of the mutated statement will be incorrect.

As an example, Figure 3 shows the function `Mid` with a mutant that replaces the relational operator $<$ with $<=$ on statement 5. The path expression for statement 5 is taken from the true branch of statement 2, and the false branch of statement 3, giving $(Y < Z) \wedge (X \geq Y)$. Note that in this case, $C_R = C_r$. The necessity constraint for the mutant requires that the two predicates evaluate to different results, that is, $((X < Z) \neq (X \leq Z))$. CBT tries to generate a test case t by satisfying a constraint system, which can be either a reachability constraint (a path expression), a necessity constraint, or a conjunction of a path expression and a necessity constraint. t must be in the union of D_R and D_n, i.e., $t \in D_R \cup D_n$. Of course, CBT is not always successful, even if the constraint system is satisfiable.

2.2 Constraints and Detecting Equivalent Mutants

In Section 1.1, an equivalent mutant was said to have the same functional behavior as the original program. However, the term "the same functional behavior" was not formally defined. Now equivalent mutants are formally defined in terms of inputs and outputs.

Definition: Let P be a program, M a mutated program of P, and $P(t)$ and $M(t)$ be the outputs of P and M on t. Then M is an *equivalent mutant program* of P iff $P(t) = M(t)$ for all t, $t \in D$.

The above definition says that if a mutant is functionally equivalent to the original program, it

```
        FUNCTION Mid (X, Y, Z : Integer)
                     RETURN Integer IS
        MidVal :  Integer;
        BEGIN
1          MidVal := Z;
2          IF (Y < Z) THEN
3            IF (X < Y) THEN
4              MidVal := Y;
5            ELSE IF (X < Z) THEN
    Δ        ELSE IF (X <= Z) THEN
6              MidVal := X;
7            END IF;
8          ELSE
9            IF (X > Y) THEN
10             MidVal := Y;
11           ELSE IF (X > Z) THEN
12             MidVal := X;
13           END IF;
14         END IF;
15         RETURN (MidVal);
16      END IF;
```

Figure 3: **Function Mid**

is impossible to find any test data to kill the mutant, that is:

$$\neg(\exists t|_{t \in D} \bullet P(t) \neq M(t)) \rightleftharpoons \forall t|_{t \in D} \bullet P(t) = M(t)$$

To support efforts in automatically detecting equivalent mutants, the following three theorems are stated and proved. These proofs are based on the definition of an equivalent mutant, descriptions of input domains, and the facts given in Section 2.1. These theorems are not deep; the essential concepts are actually encoded in the definitions.

Theorem_1: Let D_r be the domain in which test cases satisfy the reachability condition (C_r) for a mutant M. If C_r is infeasible (D_r is empty) then M is equivalent. That is, $D_r = \emptyset \Rightarrow M$ is equivalent.

Proof of Theorem_1:

1. M is equivalent $\Leftrightarrow D_s = \emptyset$	— Definition, Fact 1
2. $D_r \cap D_n \supseteq D_s$	— Fact 3
3. $D_r = \emptyset \Rightarrow D_s = \emptyset$	— Rules of sets, 2
4. $D_r = \emptyset \Rightarrow M$ is equivalent	— Substitution of 1 in 3

Theorem_2: Let D_n be the domain in which test cases satisfy the necessity condition (C_n) for a mutant M. If C_n is infeasible (D_n is empty) then M is equivalent. That is, $D_n = \emptyset \Rightarrow M$ is equivalent.

Proof of Theorem_2:

1. M is equivalent $\Leftrightarrow D_s = \emptyset$ — Definition, Fact 1
2. $D_r \cap D_n \supseteq D_s$ — Fact 3
3. $D_n = \emptyset \Rightarrow D_s = \emptyset$ — Rules of sets, 2
4. $D_n = \emptyset \Rightarrow M$ is equivalent — Substitution of 1 in 3

Theorem_3: Let D_r be the domain in which test cases satisfy the reachability condition (C_r) for a mutant M, and let D_n be the domain in which test cases satisfy the necessity condition (C_n) for M. If $C_r \wedge C_n$ is infeasible ($D_r \cap D_n$ is empty) then M is equivalent.

Proof of Theorem_3:

1. M is equivalent $\Leftrightarrow D_s = \emptyset$ — Definition, Fact 1
2. $D_r \cap D_n \supseteq D_s$ — Fact 3
3. $D_r \cap D_n = \emptyset \Rightarrow M$ is equivalent — Substitution of 1 in 2

If the weaker reachability condition (C_R) is used instead of the reachability condition (C_r), since $C_r \Rightarrow C_R$, the following two claims can easily be derived:

- $D_R = \emptyset \Rightarrow M$ is equivalent.
- $D_R \cap D_n = \emptyset \Rightarrow M$ is equivalent.

Proof of the Derivations:

1. $D_r = \emptyset \Rightarrow M$ is equivalent — Theorem_1
2. $D_R \supseteq D_r$ — Fact 4
3. $D_R = \emptyset \Rightarrow D_r = \emptyset$ — Rules of sets, 2
4. $D_R = \emptyset \Rightarrow M$ is equivalent — Transition of implication, 1, 3
5. $D_r \cap D_n = \emptyset \Rightarrow M$ is equivalent — Theorem_3
6. $D_R \cap D_n = \emptyset \Rightarrow D_r \cap D_n = \emptyset$ — Rules of sets, 2
7. $D_R \cap D_n = \emptyset \Rightarrow M$ is equivalent — Transition of implication, 5, 6

Section 2.1 mentions that CBT uses *path expression constraint systems* to represent reachability conditions (the weaker conditions) and *necessity constraint systems* to represent necessity conditions. So the following statements are true:

- If a path expression constraint system (C_R) for a statement modified by a mutant M is infeasible, then the set of test cases (D_R) that can kill M is empty – implying that M can never be killed. So M is equivalent.
- If a necessity constraint system (C_n) for a mutant M is infeasible, then the set of test cases (D_n) that can kill M is empty — implying that M can never be killed. So M is equivalent.

- If a constraint system that is a conjunction of a path expression constraint system and a necessity constraint system ($C_R \wedge C_n$) is infeasible, then the set of test cases ($D_R \cap D_n$) that can kill M is empty – implying that M can never be killed. So M is equivalent.

To decide if a constraint system is infeasible, there must be a contradiction in the constraint system itself. For example, the constraint system $(x > 0) \wedge (x < 0)$ is a contradiction, because x can never be assigned a value that is both greater than 0 and less than 0. If a mutant M has the above constraint as a path expression associated with it, then M is equivalent.

So far, the problem of detecting equivalent mutants has been translated to the problem of recognizing contradictions in mathematical constraint systems. Test data generation uses constraints to generate test cases, while equivalent mutant detection uses constraints to detect equivalent mutants.

2.3 Constraint Representation

Before describing how to use constraints to detect equivalent mutants, let us describe how constraints are represented in CBT. The basic component of a constraint is an algebraic expression, which is composed of variables, parentheses, and programming language operators. Expressions are taken directly from the test program and come from right-hand sides of assignment statements, predicates within decision statements, etc. A constraint is a pair of algebraic expressions related by one of the conditional operators $\{>, \geq, <, \leq, =, \neq\}$. Constraints evaluate to one of the Boolean values TRUE or FALSE and can be modified by the negation operator NOT ($\neg$). A *clause* is a list of constraints connected by the logical operators AND ($\wedge$) and OR ($\vee$). A *conjunctive clause* uses only the logical AND. All constraints are kept in *disjunctive normal form* (DNF), which is a list of conjunctive clauses connected by logical ORs. For example, $(x > 0)$ represents a constraint, $(x > 0) \wedge (y < 0)$ is a conjunctive clause, and $((x > 0) \wedge (y < 0)) \vee (z = 0)$ is a disjunctive formula.

In CBT, a DNF formula is referred to as a *constraint system*. DNF is used during two steps. In constraint generation, each conjunctive clause within a path expression represents a unique path to a statement. During constraint satisfaction, only one conjunctive clause needs to be satisfied. Constraints are originally created using the variables that occur in the program text. Unfortunately, this includes variables that are "internal" to the program, that is, variables that are not given values as part of the test case. For test case generation, symbolic evaluation [Kin76, Off91] is used to rewrite the variables to be in terms of input variables.

2.4 Strategies for Detecting Equivalent Mutants

The general strategy for detecting equivalent mutants is to find contradictions in the constraint systems. Because recognizing infeasible constraint systems is generally undecidable, the problem cannot be completely solved algorithmically. However, equivalent mutants are currently detected by hand, which means that partial solutions are potentially valuable.

One approach for detecting infeasible constraint systems is to apply off-the-shelf general theorem provers. Although this approach was considered, it was rejected for two reasons. First, a general purpose theorem prover would provide much more than is needed. Second, the difficulty of integrating such a system with already-existing software did not seem to be worth the effort for this research. Of course, if these results were to be used in a production system, a special purpose theorem prover might worth the effort and probably should be used.

This general strategy is applied through a collection of special case analysis techniques and heuristics to recognize infeasible constraint systems. We focus on cases that occur the most in mutation, based on the observation that mutants differ from their original programs in small, well defined ways. Section 3.2 presents empirical results that measure how well these strategies work.

This paper describes and evaluates three broad strategies that attempt to recognize infeasible constraint systems. They are negation, constraint splitting and constants comparison.

2.4.1 Negation

Definition 1: Constraint C1 is the *negation* of C2 iff the domains they describe:

- are non-overlapping, and
- cover the entire domain of the variables in C1 and C2.

To recognize infeasible constraint systems, we concentrate on constraint systems that are non-overlapping but not necessarily domain covering. The notion of partial negation is used to loosen the restriction of covering the entire domain in negation.

Definition 2: Constraint C1 is a *partial negation* of C2 iff the domains they describe:

- are non-overlapping, and
- do not cover the entire domain.

Definition 3: Two constraints are *semantically equal* if they describe the same domain.

Definition 4: Two constraints are *syntactically equal* if they describe the same domain and also have the same string of symbols.

Clearly, two syntactically equal constraints are also semantically equal.

Examples:

- Let A be the constraint $x > 1$ and B be the constraint $x \leq 1$. Then A is the *negation* of B, and B is the *negation* of A (negation is commutative). Both constraints cannot be satisfied at the same time (their domains are non-overlapping), but the domain of x that makes the two constraints TRUE cover the entire domain of x.

- Let A be the constraint $x > 1$ and B be the constraint $x < 1$. Then A is a *partial negation* of B, and B is a *partial negation* of A. Both constraints cannot be satisfied at the same time, but the domain of x that makes the two constraints TRUE does not cover the entire domain of x.

- Suppose constraint A is $x > 0$ and constraint B is $x > 0$. Then A and B are syntactically equal. Thus, A and B are semantically equal.

- Let x be an integer variable, A be the constraint $x > 0$ and B be the constraint $x \geq 1$. A and B are not syntactically equal, but they are semantically equal.

The negation strategy is the basic technique used to recognize infeasible constraints. Given two constraints, *negation* or *partial negation* is first used to rewrite one of the constraints, then these two constraints are compared. If they are syntactically equal, the constraints conflict, and the constraint system is infeasible. So, a mutant with this infeasible constraint system is equivalent.

For example, assume two constraints A and B, where A is $(x + y) > z$ and B is $(x + y) \leq z$. The negation of A is $(x + y) \leq z$, denoted A'. Since A' and B are syntactically equal, A and B conflict. The negation algorithm is given in Figure 4; it uses the negation and partial negation functions defined in Table 3.

```
   algorithm:     Negation (A, B)
   precondition:  A and B are properly initialized constraints.
   postcondition: Returns conflict if A and B conflict, no-conflict otherwise.

BEGIN
    -- Use Table 3 (Negation table) to negate A.
    neg-A = Negate(A)
    IF (neg-A syntactically equals B)
        RETURN conflict
    ELSE
        IF (the relation operator in A is one of {>,<,=})
            -- Use Table 3 (Negation table) to negate B.
            partneg-A = PartialNegate1(A)
            IF (partneg1-A syntactically equals B)
                RETURN conflict
            ELSE
                partneg2-A = PartialNegate2(A)
                IF (partneg2-A syntactically equals B)
                    RETURN conflict
                ELSE
                    RETURN no-conflict
                END IF
            END IF
        END IF
    END IF
END Negation
```

Figure 4: **The Negation Algorithm – Decides if Two Constraints Conflict with Each Other.**

Constraint C	**Negation of C**	**Partial Negation of C**	
		Partial Negation1	**Partial Negation2**
expr1 $>$ expr2	expr1 $\leq$ expr2	expr1 $<$ expr2	expr1 $=$ expr2
expr1 $\geq$ expr2	expr1 $<$ expr2	—	—
expr1 $<$ expr2	expr1 $\geq$ expr2	expr1 $>$ expr2	expr1 $=$ expr2
expr1 $\leq$ expr2	expr1 $>$ expr2	—	—
expr1 $=$ expr2	expr1 $\neq$ expr2	expr1 $>$ expr2	expr1 $<$ expr2
expr1 $\neq$ expr2	expr1 $=$ expr2	—	—
True	False	—	—
False	True	—	—

Table 3: **Negation and Partial Negation**

2.4.2 Constraint splitting

Constraint splitting is also used to recognize infeasible constraints. A commonly occurring case is a necessity constraint such as $(x + y) > 0$, together with a path expression such as $(x < 0) \wedge (y < 0)$. The negation strategy cannot recognize that these conflict.

To detect such conflicts, a strategy called *constraint splitting* is used. Given two constraints (say C and D), two new constraints (say A and B) are generated such that $C \Rightarrow A \vee B$. Then A and B are compared with D. The following proves that if both A and B conflict with D, then C conflicts with D, thus showing the correctness of the constraint splitting strategy.

$$\begin{array}{lll} & C \Rightarrow A \vee B & \\ \Leftrightarrow & \neg C \vee (A \vee B) & \text{— implication} \\ \Leftrightarrow & (A \vee B) \vee \neg C & \text{— commutativity} \\ \Leftrightarrow & \neg\neg(A \vee B) \vee \neg C & \text{— logical negation} \\ \Leftrightarrow & \neg(\neg A \wedge \neg B) \vee \neg C & \text{— DeMorgan's law} \\ \Leftrightarrow & \neg A \wedge \neg B \Rightarrow \neg C & \text{— implication} \end{array}$$

By showing that A and B conflict with D, that is, $\neg A \wedge \neg B \wedge D$, and using the above proof, $\neg A \wedge \neg B \Rightarrow \neg C$, we are sure that C conflicts with D, that is, $\neg C \wedge D$. The following proves this:

From $\neg A \wedge \neg B \wedge D$, $\neg A \wedge \neg B \Rightarrow \neg C$ Infer $\neg C \wedge D$

1	$\neg A \wedge \neg B \wedge D$	premise
2	$\neg A \wedge \neg B$	property of And, 1
3	$\neg A \wedge \neg B \Rightarrow \neg C$	premise
4	$\neg C$	implication eliminating, 2, 3
5	D	property of And, 1
6	$\neg C \wedge D$	property of And, 4, 5

For the *constraint splitting* strategy, the cases shown in Table 4 are analyzed. Note that for most of these, A and B are weaker than C, but it is usually easier to decide if A or B conflicts with D. The algorithm for performing constraint splitting is given in Figure 5.

2.4.3 Constants comparison

A third strategy uses a property that is common in constraints created for test case generation. The property is *grounding*, where both constraints have the format (V rop K), where V is a variable, rop is a relational operator, and K is a constant. In addition, the variables in both constraints must be the same.

Let A be the constraint (X rop1 K1), and B be the constraint (X rop2 K2). By evaluating the two constants and relational operators, we can often decide whether A conflicts with B. This strategy is called *constants comparison*. Table 5 shows the cases analyzed for the *constants comparison* strategy. The first and the second columns are the two given constraints. The third column is a predicate on constants $k1$ and $k2$, which is used to decide whether the two constrains conflict.

Original Constraint		**New Constraint1**		**New Constraint2**
$(x+y) > 0$	$\Rightarrow$	$x > 0$	$\vee$	$y > 0$
$(x+y) \geq 0$	$\Rightarrow$	$x \geq 0$	$\vee$	$y \geq 0$
$(x+y) < 0$	$\Rightarrow$	$x < 0$	$\vee$	$y < 0$
$(x+y) \leq 0$	$\Rightarrow$	$x \leq 0$	$\vee$	$y \leq 0$
$(x+y) = 0$	$\Rightarrow$	$x \leq 0$	$\vee$	$y \leq 0$
$(x+y) \neq 0$	$\Rightarrow$	$x \neq \Leftrightarrow y$		
$(x \Leftrightarrow y) > 0$	$\Rightarrow$	$x > 0$	$\vee$	$y < 0$
$(x \Leftrightarrow y) \geq 0$	$\Rightarrow$	$x \geq 0$	$\vee$	$y \leq 0$
$(x \Leftrightarrow y) < 0$	$\Rightarrow$	$x < 0$	$\vee$	$y > 0$
$(x \Leftrightarrow y) \leq 0$	$\Rightarrow$	$x \leq 0$	$\vee$	$y \geq 0$
$(x \Leftrightarrow y) = 0$	$\Rightarrow$	$x \leq 0$	$\vee$	$y \geq 0$
$(x \Leftrightarrow y) \neq 0$	$\Rightarrow$	$x \neq y$		
$(x \times y) > 0$	$\Rightarrow$	$x > 0 \wedge y > 0$	$\vee$	$x < 0 \wedge y < 0$
$(x \times y) \geq 0$	$\Rightarrow$	$x \geq 0 \wedge y \geq 0$	$\vee$	$x \leq 0 \wedge y \leq 0$
$(x \times y) < 0$	$\Rightarrow$	$x > 0 \wedge y < 0$	$\vee$	$x < 0 \wedge y > 0$
$(x \times y) \leq 0$	$\Rightarrow$	$x \geq 0 \wedge y \leq 0$	$\vee$	$x \leq 0 \wedge y \geq 0$
$(x \times y) = 0$	$\Rightarrow$	$x = 0$	$\vee$	$y = 0$
$(x \times y) \neq 0$	$\Rightarrow$	$x \neq 0 \wedge y \neq 0$		
$(x \div y) > 0$	$\Rightarrow$	$x > 0 \wedge y > 0$	$\vee$	$x < 0 \wedge y < 0$
$(x \div y) \geq 0$	$\Rightarrow$	$x \geq 0 \wedge y > 0$	$\vee$	$x \leq 0 \wedge y < 0$
$(x \div y) < 0$	$\Rightarrow$	$x > 0 \wedge y < 0$	$\vee$	$x < 0 \wedge y > 0$
$(x \div y) \leq 0$	$\Rightarrow$	$x \geq 0 \wedge y < 0$	$\vee$	$x \leq 0 \wedge y > 0$
$(x \div y) = 0$	$\Rightarrow$	$x = 0$		
$(x \div y) \neq 0$	$\Rightarrow$	$x \neq 0$		

Table 4: **Constraint Splitting Cases Analysis**

```
   algorithm:     SplitConstraint (NecConst, PEConst)
   precondition:  NecConst and PEConst are properly initialized constraints.
   postcondition: Returns conflict if NecConst and PEConst conflict, no-conflict otherwise.


BEGIN
    -- V1 and V2 are variables, K is a constant, aop is an arithmetic operator,
    -- and rop is a relational operator.
    IF (the format of NecConst is not ((V1 aop V2) rop K))
        RETURN no-conflict
    ELSE
        -- Use Table 4 (Constraint Splitting table) to split NecConst.
        A = NewConstraint1(NecConst)
        B = NewConstraint2(NecConst)
    END IF
    IF (Negation (A, PEConst)) AND (Negation (B, PEConst))
        RETURN conflict
    ELSE
        IF (CompareConstraints (A, PEConst)) AND (CompareConstraints (B, PEConst))
            RETURN conflict
        ELSE
            RETURN no-conflict
        END IF
    END IF
END SplitConstraint
```

Figure 5: **The SplitConstraint Algorithm – Splits a Necessity Constraint Into Two Parts and Decides if They Both Conflict With the PE.**

Note that it does not always have such a predicate available. The last column is the conclusion of whether the two constraints conflict. The word "pred" stands for "the predicate holds"; the predicate is in column 3. The letter "T" stands for "True", which means the two given constraints conflict; the letter "F" stands for "False", which means the two given constraints do not conflict.

Given the two constraints $(x > 1)$ and $(x < 0)$, if the negation strategy is used, the first constraint $(x > 1)$ can be partially negated or negated to be $(x < 1)$ or $(x \leq 1)$, but neither $(x < 1)$ nor $(x \leq 1)$ is syntactically equal to $(x < 0)$, so we cannot tell that they conflict. Yet constants comparison can be used to show that they conflict.

To expand the use of constants comparison, if a constraint has the format (V aop K1) rop K2, it is rewritten as V rop (K2 $\overline{aop}$ K1), such that (V aop K1) rop K2 $\Leftrightarrow$ V rop (K2 $\overline{aop}$ K1), where V is a variable, aop is an arithmetic operator, $\overline{aop}$ is the mathematical inverse operation of aop, rop is a relational operator, and K1 and K2 are constants. The algorithm for performing constants comparison is given in Figure 6.

3 Empirical Evaluation

The mathematical basis for these strategies allows it to be proved that they can successfully detect at least some equivalent mutants. But this leaves the question of how well they will work – how many equivalent mutants they will detect. This section describes a proof-of-concept tool that was built to demonstrate that the technique could be practically applied, and then describes some empirical results from using the tool.

3.1 A Constraint-Based Equivalence Detection Tool

The proof-of-concept tool, Equivalencer, is integrated with Godzilla, a test data generator within the Mothra mutation tool set. Although the CBT technique is language independent, Mothra works with Fortran 77 programs. Equivalencer was implemented in the programming language *C* on a Sun Sparc classic workstation running the SunOS 4.1.3 operating system. Equivalencer contains more than 2000 lines of executable source code and also uses several Mothra and Godzilla library packages.

Figure 7 graphically shows the general execution flow of Equivalencer. For each mutant created for a program, each of the three equivalence detection strategies described in Section 2.4 is applied in turn.

This section first describes assertion constraints, which affect the design and implementation, and how they are inserted into a program under test. Then the architectural design for the tool *Equivalencer* is given.

3.1.1 Inserting Assertion Constraints

In Equivalencer, assertion constraints are used to help detect equivalent mutants. Assertions are constraints that a user inserts into a test program to manually restrict the input domain of some variables. They could be preconditions to the program, predicates on specific statements, or predicates that apply to an entire function or program.

There are two kinds of variables in a program or a function: parameter variables and internal variables. The assertions on the parameter variables can usually only be derived by a human

Constraint A	**Constraint B**	**Predicate (pred)**	**Conclusion**
$x > k1$	$x > k2$	—	F
$x > k1$	$x \geq k2$	—	F
$x > k1$	$x < k2$	$k1 \geq k2 \Leftrightarrow 1$	if pred T, else F
$x > k1$	$x \leq k2$	$k1 \geq k2$	if pred T, else F
$x > k1$	$x = k2$	$k1 \geq k2$	if pred T, else F
$x > k1$	$x \neq k2$	—	F
$x \geq k1$	$x > k2$	—	F
$x \geq k1$	$x \geq k2$	—	F
$x \geq k1$	$x < k2$	$k1 \geq k2$	if pred T, else F
$x \geq k1$	$x \leq k2$	$k1 > k2$	if pred T, else F
$x \geq k1$	$x = k2$	$k1 > k2$	if pred T, else F
$x \geq k1$	$x \neq k2$	—	F
$x < k1$	$x > k2$	$k1 \leq k2 + 1$	if pred T, else F
$x < k1$	$x \geq k2$	$k1 \leq k2$	if pred T, else F
$x < k1$	$x < k2$	—	F
$x < k1$	$x \leq k2$	—	F
$x < k1$	$x = k2$	$k1 \leq k2$	if pred T, else F
$x < k1$	$x \neq k2$	—	F
$x \leq k1$	$x > k2$	$k1 \leq k2$	if pred T, else F
$x \leq k1$	$x \geq k2$	$k1 < k2$	if pred T, else F
$x \leq k1$	$x < k2$	—	F
$x \leq k1$	$x \leq k2$	—	F
$x \leq k1$	$x = k2$	$k1 < k2$	if pred T, else F
$x \leq k1$	$x \neq k2$	—	F
$x = k1$	$x > k2$	$k1 \leq k2$	if pred T, else F
$x = k1$	$x \geq k2$	$k1 < k2$	if pred T, else F
$x = k1$	$x < k2$	$k1 \geq k2$	if pred T, else F
$x = k1$	$x \leq k2$	$k1 > k2$	if pred T, else F
$x = k1$	$x = k2$	$k1 \neq k2$	if pred T, else F
$x = k1$	$x \neq k2$	$k1 = k2$	if pred T, else F
$x \neq k1$	$x > k2$	—	F
$x \neq k1$	$x \geq k2$	—	F
$x \neq k1$	$x < k2$	—	F
$x \neq k1$	$x \leq k2$	—	F
$x \neq k1$	$x = k2$	$k1 = k2$	if pred T, else F
$x \neq k1$	$x \neq k2$	—	F

Table 5: **Constants Comparison Cases Analysis**

```
algorithm:      CompareConstants (A, B)
precondition:   A and B are properly initialized constraints.
postcondition:  Returns conflict if A and B conflict, no-conflict otherwise.

BEGIN
    -- V is a variable, K, K1, and K2 are constants,
    -- rop is a relational operator, and aop is an arithmetic operator.
    IF (the format of A is (V rop K))
        keep the format the same
    ELSE IF (the format of A is (K rop V))
        modify the format to (V $\overline{rop}$ K)
    ELSE IF (the format of A is ((V aop K1) rop K2))
        modify the format to (V rop (K2 $\overline{aop}$ K1))
    ELSE IF (the format of A is (K1 rop (V aop K2)))
        modify the format to (V $\overline{rop}$ (K1 $\overline{aop}$ K2))
    ELSE
        RETURN no-conflict
    END IF
    IF (the format of B is (V rop K))
        keep the format the same
    ELSE IF (the format of B is (K rop V))
        modify the format to (V $\overline{rop}$ K)
    ELSE IF (the format of B is ((V aop K1) rop K2))
        modify the format to (V rop (K2 $\overline{aop}$ K1))
    ELSE IF (the format of B is (K1 rop (V aop K2)))
        modify the format to (V $\overline{rop}$ (K1 $\overline{aop}$ K2))
    ELSE
        RETURN no-conflict
    END IF
    IF (V in A and B are not the same)
        RETURN no-conflict
    END IF
    -- Use Table 5 (Constants Comparison table) to check A and B.
    IF (ConstantsComparison(A, B) == True))
        RETURN conflict
    ELSE
        RETURN no-conflict
    END IF
END CompareConstants
```

Figure 6: **The CompareConstants Algorithm – Compares Two Constraints Using the Principle of Grounding to See If They Conflict.**

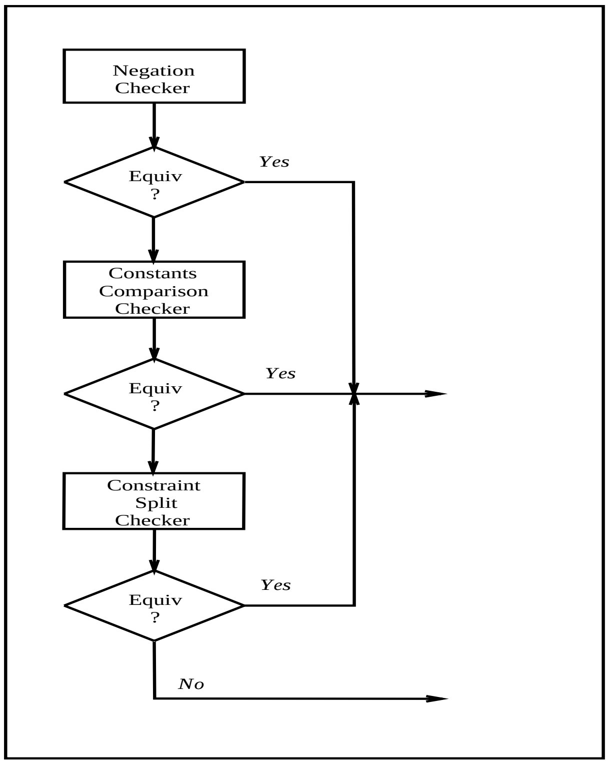

Figure 7: **Infeasible Constraint Checker**

(semantically). These are often part of the specifications, that is, preconditions. The assertions on the internal variables can sometimes be derived automatically (syntactically). Three kinds of assertions in the programs are used in this paper.

1. Assertions on parameter variables. These assertions usually encode preconditions.
2. Assertions on internal variables that could be derived automatically. There are known techniques for doing this, such as slicing [Wei84] and control flow analysis [FL88]. Because the focus of this research was on detecting equivalent mutants, these assertions were generated by hand and inserted as assertion constraints.
3. Assertions on internal variables that could not easily be derived automatically.

When Godzilla generates a constraint that contains arrays, it takes a conservative approach that does not provide array index expressions associated with array references. For example, a constraint system such as $A(i) \geq 0 \wedge A(j) < 0$ will be generated as $A() \geq 0 \wedge A() < 0$. Depending on the negation strategy, this constraint system could be recognized as having a contradiction in it, which is incorrect. To avoid this, Equivalencer skips checking constraints with arrays, except in the case described below. This is a severe limitation on the experimental proof-of-concept tool that should be addressed in a practical implementation.

Constraints with arrays are used to recognize contradiction only if this constraint is an assertion constraint. This is because it is assured that an assertion constraint involving an array will apply to every element in the array. For example, an assertion constraint is $A() \geq 0$, which means every element in array $A()$ is greater than or equal to 0. If there is a necessity constraint, say $A() < 0$, it is definite that this assertion conflicts with this necessity constraint. To take advantage of this, a routine was added to check whether assertion constraints conflict with other constraints, such as necessity and path expression constraints. In this routine, Equivalencer works on array constraints. This routine is referred to as *array-extension*.

3.1.2 Architectural Design

Figure 8 shows the execution flow of Equivalencer. The first step is the initialization. Equivalencer opens all the files that are needed and brings the data into memory. In the second step, Equivalencer consults the *Failure Information* offered by Godzilla. In simple cases, the test data generator is able to recognize equivalent mutants. If the information says the mutant being checked is equivalent, it outputs an equivalent message and exits, otherwise Equivalencer goes to the next step. Next Equivalencer gets the *Path Expression* (**pe**) (from Godzilla) and *Assertion Constraints* (**assertion**), and combines them to form a completed path expression (**pathexpr**). Then it checks for infeasible constraints in **pathexpr** by using the negation, constants comparison, and constraint splitting strategies. If the constraints are infeasible, it outputs the equivalent message, otherwise it goes to the next step. Equivalencer gets the *Necessity Constraint* (**cnst**) if it is available, otherwise it outputs a message indicating that fact. If **cnst** is available then Equivalencer checks whether **cnst** is infeasible by using the negation, constraint splitting, and constants comparison strategies. If it is infeasible, Equivalencer outputs an equivalence message. If it is not, Equivalencer goes to the next step. After combining **cnst** and **pathexpr**, Equivalencer checks for a contradiction in the combination by using the three strategies. If a contradiction is found, it outputs an equivalence message. If it is not found, the array-extension checking routine is applied to assertion constraints against **cnst** and against **pathexpr**. If a conflict is found, an equivalence message is output. If none of these steps find that the mutant is equivalent, Equivalencer outputs a message indicating that fact. After outputting the messages, Equivalencer checks whether there are more mutants. If not, it is finished and it terminates.

As a proof-of-concept prototype, efficiency was not a high priority when Equivalencer was built. The tool compares each **pathexpr** from each statement with each **cnst** that is defined at that statement. So the running time is constant in the number of necessity constraints, which is on the order of the number of mutants. As shown in a previous paper [OLR+96], the number of mutants is $O(S^2)$, where S is the number of statements in the program. Because the focus of this study was a proof-of-concept, execution time information was not precisely monitored, but when compared with hand analysis of mutants, the execution time of this tool was trivial.

3.2 Empirical Results

A controlled study was carried out using Equivalencer to determine equivalent mutants in 11 Fortran 77 programs that cover a range of applications. These programs range in size from about 11 to 30 executable statements and have from about 180 to 3000 mutants. Because the speed of these equivalence procedures is infinitesimal when compared to the speed of mutation testing, the size of the programs is relatively unimportant. However, each program had to be analyzed by hand to determine the true number of equivalent mutants. Equivalencer's effectiveness has been compared based on the percentage of equivalent mutants that it detected. The equivalent mutants for these programs have been analyzed by several researchers.

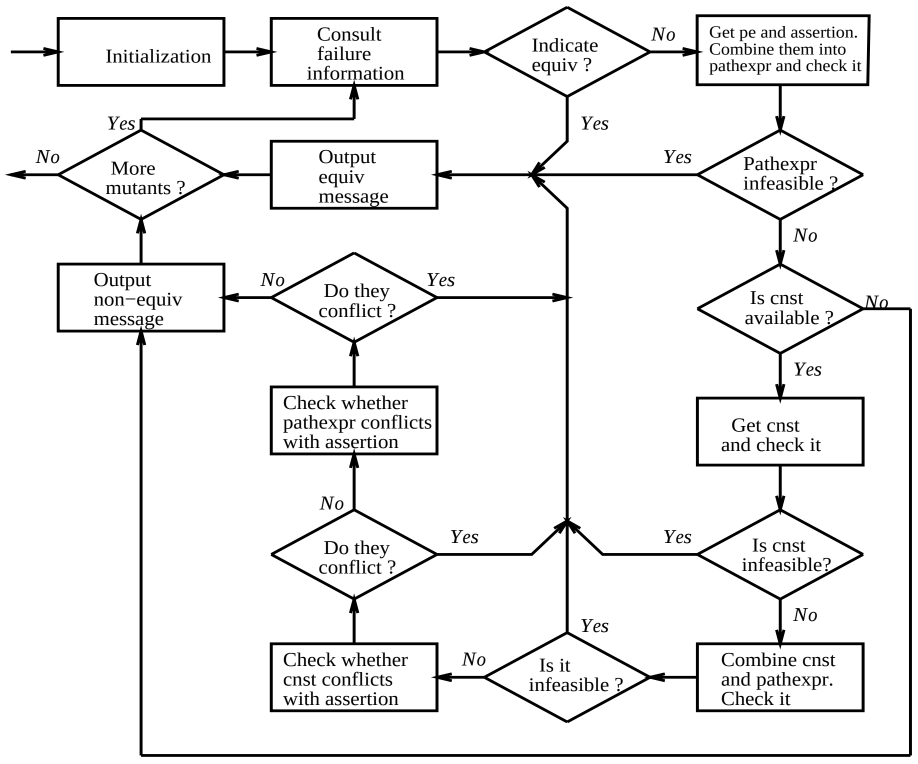

Figure 8: **Equivalencer Execution Diagram**

Program	Statements	Mutants	Equivalent	Equiv. Detected	Percent Detected
Bsearch	20	299	27	19	70.37
Bub	11	338	35	24	68.57
Cal	29	3010	236	37	15.67
Euclid	11	196	24	18	75.00
Find	28	1022	75	63	84.00
Insert	14	460	46	32	69.57
Mid	16	183	13	3	23.08
Pat	17	513	61	29	47.54
Quad	10	359	31	4	12.90
Trityp	28	951	109	80	73.39
Warshall	11	305	35	22	62.86
Total/Avg	185	7636	695	331	47.63

Table 6: **Equivalent Mutants Detected**

3.2.1 Equivalence detection

The procedure was relatively straightforward:

1. The lists of equivalent mutants were gathered from previous research.
2. For each program, Mothra was used to generate all mutants, and Godzilla was used to generate the path expression (reachability) constraints and necessity constraints.
3. Assertion constraints were generated for programs that included hand-inserted assertions. These assertions were created by the first author.
4. Equivalencer was run to see how many equivalent mutants could be detected.
5. The mutants that were marked equivalent were compared with the hand-generated lists.

The data are displayed in Table 6. The number of executable statements, mutants, equivalent mutants, equivalent mutants detected, and the percent of equivalent mutants detected for each program is shown. The bottom line shows the total number of statements, mutants, and equivalent mutants in the programs, and the average number detected. The average detected is calculated from the totals, not the percent detected for each program (that is, it is not a weighted average).

3.2.2 Discussion

Although Equivalencer was able to detect roughly half of the equivalent mutants on average, the percent found for each program varied greatly. This is due to a variety of factors, perhaps most importantly being limitations of the tools Equivalencer is based on.

Godzilla treats arrays as single data items and references to array elements are treated as references to the entire array. For this reason, many constraints involving array references could not be used to determine equivalence. An example of this is the statement A(5) = 0. ¿From this definition, the fact that the fifth element of A is set to zero can be determined, and a later mutant that could only be killed if A(5) was not 0 is equivalent. If elements of an array could be treated as individual data items, the constraint-based technique could be used to detect more equivalent mutants. This problem is particularly important for the `Cal` program above, for which only 15% of the equivalent mutants were detected. Recent results with a "dynamic-domain reduction" approach to generating tests [OJP94] indicate that arrays can be handled within constraints as individual elements – perhaps allowing many more equivalent mutants to be detected.

Godzilla associates every variable in each constraint with a statement number. Usually, the statement number for a variable in a constraint is the number of the statement associated with the constraint. For example, a constraint $A < 0$ in statement 5 will be expressed as $A.5 < 0$. But if a variable's value has not changed from the previous statement, the same statement number for the variable might be used. For example, using the constraint in the example above, if A has not changed from statement 4, it might be shown as $A.4 < 0$ on statement 5. This affects both the handling of assertions and symbolic evaluation.

Assertions in Godzilla are handled in one of two ways. *Global assertions* are applied to all statements in the program, and *local assertions* are applied only to the statement immediately following the assertion. Global assertions are used as constraints that carry the same statement number (usually 1) to each statement. Unfortunately, Godzilla is limited to treating all assertions of a program as either global or local, thus it was necessary to gather data using global assertions separately from the data using local assertions. Th assertions increased the overall average percent detected by almost 5%.

Godzilla's symbolic evaluator presented a more formidable problem. The symbolic evaluator rewrites constraints to be solely in terms of the input parameters by symbolically evaluating the program. Also, Godzilla rewrites the statement numbers in the constraints. Initially, the statement number for each variable is the statement where the constraint appears. Godzilla rewrites a variable's statement number to a previous statement number if the variable has not been assigned a new value from the previous statement. This is called *constraints propagation* because the statement numbers are "propagated" through the constraints. Constraints propagation, especially in path expression constraints, increases Equivalencer's detection ability. Unfortunately, Godzilla has a design flaw that causes it to work incorrectly on constraints involving loops. If a constraint re-written by Godzilla is in a loop, it is correct for the first iteration, but usually not correct in the second and subsequent iterations. This caused no problems for the original purpose of test data generation, but made equivalent mutant detection more difficult. Since Equivalencer detects equivalent mutants using constraints, Equivalencer will detect incorrect equivalent mutants if the constraints are wrong. Unfortunately, fixing the design flaw for Godzilla was impractical, thus the mutants that were incorrectly identified as equivalent were removed by hand.

A previous automatic equivalent mutant detector was presented by Offutt and Craft [OC94]. It used compiler optimization techniques to identify equivalent mutants of the original program. When run on the same programs, the constraint-based technique detected almost 5 times more equivalent mutants than the compiler optimization techniques.

A test suite of eleven Fortran-77 programs was used for these studies. It cannot be claimed that these programs represent a statistically valid sample of programs. There is no generally accepted way to choose such a sample of programs, and this paper does not attempt to solve that problem. However, the programs (functions) were taken from the literature and chosen to represent different types of problems to exercise the equivalent mutant detection capabilities in as wide a manner as possible.

3.2.3 Feasible Path Results

Although the focus in this research was on equivalent mutants, the same techniques can be applied to the problem of detecting infeasible paths. Specifically, a simple corollary to Theorem 1 is that if the reachability condition for a statement is infeasible, the statement is unreachable. Thus, a preliminary evaluation of using the constraint-based technique to recognize unreachable statements

Program	Unreachable	Detected	Percent Detected
Prog 1	2	1	50.00
Prog 2	1	0	0.00
Prog 3	1	1	100.00
Prog 4	1	1	100.00
Prog 5	1	1	100.00
Prog 6	1	1	100.00
Prog 7	2	2	100.00
Prog 8	3	3	100.00
Prog 9	2	0	0.00
Total/Avg	14	10	71.43

Table 7: **Unreachable Statements Detected**

was made. To do this, programs were artificially constructed so that some of their statements were not reachable, and the Equivalencer was used to try to detect the unreachability. This was done by applying the mutation operator that ensures statement coverage; the mutants are killed by a test case if and only if the test case reaches the statement. If Equivalencer found that the mutant is equivalent, that means the statement is unreachable.

Artificially creating programs with unreachable statements implies a severe limitation and potential bias, so these results should not be generalized without further research. On the other hand, the preliminary results in Table 7 are encouraging. Of the 9 programs, there were 14 unreachable paths, and the tool was able to find 10 of those. Many of the equivalent mutants that Equivalencer was not able to detect were equivalent because of computation that occurred after the mutated statement was executed; since this is not a problem for unreachable paths, it is not surprising to get better results for unreachable paths than for equivalent mutants.

4 Suggestions for Improvement and Future Work

There are several improvements that should be used in a practical tool. These are divided into two categories: recommendations for improving the software, and recommendations for improving the technique.

4.1 Improving the Software

The Equivalencer tool relied very heavily on the pre-existing Godzilla test data generator. Godzilla implements symbolic evaluation as a separate step from infeasible constraint recognition. Although symbolic evaluation helps Equivalencer detect equivalent mutants by propagating the constraints, it increases the difficulty of detection by throwing away considerable information that Equivalencer needs, such as all references to internal variables. Also, Godzilla does not propagate the assertion constraints. In future systems, the symbolic evaluation should be merged with infeasible constraint recognition. This should allow for more detection of equivalent mutants, as well as being more efficient and flexible.

4.2 Improving the Detection Techniques

Three ways are recommended to improve the techniques. One is another strategy that could recognize infeasible constraints, another is to have better constraints, and the third one is to analyze the execution after the mutated statement.

Three strategies for recognizing infeasible constraints were presented in Section 2.4. Another potential strategy is the following. Assume a constraint X. Suppose other constraints can be found, say $C1, C2, ..., Cn$, such that $X \wedge C1 \wedge C2 \wedge ... \wedge Cn \Rightarrow Y$, and $C1, C2, ..., Cn$ are `TRUE`. If it can be proved that $\neg Y$ holds, then it can be said that $\neg X$ holds, which means that X is an infeasible constraint. An analysis of the programs studied uncovered only one equivalent mutant that could be detected using this strategy. Thus, this strategy was not implemented, but it is possible that it could detect more equivalent mutants in other programs.

One limitation has to do with array constraints generation. Right now, Godzilla generates array constraints without indexes, which is a safe approach when indexes are variables. But programs often contain array constraints with indexes that are constants. Analysis of the program `Cal` shows that if array constraints with constant-indexes could be used, Equivalencer would detect 69 more equivalent mutants, which would increase the detection percentage from 15.67% to 44.92% in program `Cal`.

Another thought on improving the constraints is that of using humans to help with difficult constraints. We imagine a system that interacts with the tester to get help with difficult constraints. Although this would require more work from the tester than automatic detection, it is still less than requiring hand recognition of equivalent mutants. That is, it is easier for a human to recognize infeasible constraints than equivalent mutants.

This research only analyzed the execution up to and including the mutated statement. If execution after the mutated statement could be analyzed, then many more equivalent mutants could be recognized. Analysis of the execution after the mutated statements requires analysis of sufficiency conditions, which are not generated by Godzilla, but we see no reason why this could not be done.

5 Conclusions

This paper presents a partial solution to the problem of equivalent mutant detection, and to the feasible path problem, and shows the specific relationship between them. The solution is proposed, specific algorithms are presented, and a proof-of-concept experimental tool is described. The results show that the approach is an effective partial solution to this problem. It is also shown that this technique can be applied to the feasible path problem, possibly giving better results than with equivalent mutants. This general technique is generalizable to all instances of what is defined as the feasible test problem, and can thus be used to support branch coverage techniques and data flow testing.

By using the CBT technique, Equivalencer is able to automatically detect a significant percentage for most programs, although it is not possible to detect all equivalent mutants. In the experiments, the detection percentage is over 60% for 7 of the 11 programs and the average detection percentage over all programs is over 45% (see Table 6 in Section 3.2). With appropriate extensions, the detection percentages could be even higher. Also the detection time is reasonably fast, even in this implementation, which was not optimized for speed. Compared with running every test case against those equivalent mutants, the time saving is large. Compared with detecting those equivalent mutants by hand, the time saving is significant.

Previous research [OC94] reported results from using compiler optimization techniques for this problem. The ideas in this paper are derived from that work, and it is clear that CBT is a far more effective approach.

This research is part of a long term project to provide practical, powerful automated test environments to testers, so that highly assured software can be produced at reasonable cost. We envision a system that provides almost complete automation to the tester. This type of system would allow a programmer to submit a software module, and after a few minutes of computation, respond with a set of test cases that are assured of providing the software with a very effective test, and a set of outputs that can be examined to find failures in the software. Furthermore, these input-output pairs can be used as a basis for debugging when failures are found.

References

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

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

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

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

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

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

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

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

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[GWZ94] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 80–94, Seattle WA, August 1994.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[How82] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[JBW$^+$94] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 95–107, Seattle WA, August 1994.

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

[Mor90] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[Mye79] G. Myers. *The Art of Software Testing.* John Wiley and Sons, New York NY, 1979.

[OC94] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4(3):131–154, September 1994.

[Off88] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. Technical report GIT-ICS 88/28.

[Off91] A. J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[OJP94] J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach for test data generation: Design and algorithms. Technical report ISSE-TR-94-110, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, September 1994.

[OLR$^+$96] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[UOH93] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge MA, June 1993.

[VMM91] J. M. Voas, L. Morell, and K. W. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–58, March 1991.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.