Using Constraints to Detect Equivalent Mutants*

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

August 1994

Abstract

Mutation testing is a software testing technique that is considered to be very powerful, but is very expensive to apply. One open problem in mutation testing is how to automatically detect equivalent mutant programs. Currently, equivalent mutants are detected by hand, which makes it a very expensive and time-consuming process, and restricts the use of mutation testing. This paper presents a technique that uses mathematical constraints to automatically detect equivalent mutants. A tool *Equivalencer* has been developed to demonstrate this technique, and experimental results from using this tool are presented.

*Supported by the National Science Foundation under grant CCR-93-11967.

Contents

1 Introduction

Mutation analysis is a technique for testing software. One open problem in mutation testing is how to automatically detect equivalent mutated programs. Currently, equivalent mutants are detected by hand, which it makes a very expensive and time-consuming process. This thesis describes an approach that is based on the constraint-based testing technique by recognizing infeasible constraints to automatically detect equivalent mutants, applies a collection of special case analysis and heuristics to recognize infeasible constraints, and presents a tool that implements this approach.
In the rest of this thesis, we will use following notations and definitions. If P is a program, then P(t) denotes the output value of the function computed by P on input t. The set of all inputs at which P defines a definite value is the *domain* of P. The letter D is usually used to denote the domain of a program. A *test case* is an input that belongs to D. A *test case set* T is a subset of D that contains test cases (inputs) that are used to test P under some testing criterion. We use *test data* interchangeably with *test case set*. A *failure* in a system is an observable event where the system violates its specifications. An *error* is an item of information that may produce a failure when processed by the system. A *fault* is a mechanical or algorithmic defect that will generate an error.

1.1 Software Testing

A software life cycle includes software requirements, specification, design, implementation, testing and maintenance. Software testing is one of and often the most time consuming part of the software development process [Som92]. Sommerville defines various levels at which software testing occurs. They can be identified as *unit test* (also known as a *module test*), *integration test*, *subsystem test*, *system test* and *acceptance test*. There are two testing approaches that can be applied at these levels. They are refered as "black box" or "white box" approaches. A *black box* testing approach will devise test data without any knowledge of the structure of the software under test, whereas *white box* testing will explicitly use the program structure to develop test data. Black box testing is based on the requirements and specifications, while white box testing is based on the source code. Usually, white box testing approaches are applied to the unit test level, and black box testing approaches are applied during integration test and system test.
Software testing is the process of executing a program on a set of test cases and comparing the actual results with the expected results. Its purpose is to reveal the existence of faults. With this purpose in mind, the testing criterion is often stated as "test until the software is fault-free". Unfortunately, we cannot use conventional testing strategies to guarantee that the software is fault-free [How76]. Therefore, one purpose of a software testing strategy is to specify a stopping point for testing while developing test data that is useful for finding faults.
There are several criteria that may be used to specify a stopping point for testing. Statement testing, branch testing, data-flow testing, and fault-based testing are a few examples. They are all white box testing approaches. Statement testing [Mye79] requires that the testing continues until all statements have been reached. With the branch testing criterion [Hua75], all branches need to be executed before the testing can be finished. In data-flow testing, seven distinct criteria have been defined [FW88] for data-flow; testing continues until one or all of the criteria have been satisfied. With fault-based testing techniques, test cases are designed to find specific types of faults that usually represent common programming mistakes. Testing continues until all these faults have been found. Mutation testing is a fault-based testing technique.

1.2 Mutation Analysis

Mutation is a fault-based testing technique that measures the adequacy of a set of externally created test cases [DLS78], [Ham77]. Mutation testing creates a collection of programs that have one simple syntactic change from the program under test by using mutant operators. These programs are called *mutated programs* or *mutants*. We call the program under test the *original program*. A *mutant operator* is a rule for changing the original program to create mutants. The goal of the tester is to find a test case that causes each mutant to generate an output different from that of the original program on the same test case, thereby distinguishing

```
   INTEGER FUNCTION Min (I,J)
   INTEGER I, J

1    Min = I
2    IF (J .LT. I) Min = J
3    RETURN
```

```
   INTEGER FUNCTION MinMut (I, J)
   INTEGER I, J

1  Δ1  Min = J
2      IF (J .LT. I) Min = J
3      RETURN
```

Figure 1: **Function Min and a Mutant**

```
   INTEGER FUNCTION Min (I,J)
   INTEGER I, J

1        Min = I
   Δ1    Min = J
   Δ2    Min = ABS(I)
2        IF (J .LT. I) Min = J
   Δ3    IF (J .GT. I) Min = J
   Δ4    IF (J .LT. I) TRAP
   Δ5    IF (J .LT. Min) Min = J
3        RETURN
```

Figure 2: **Function Min with Five Mutants** *(In-line Version)*

the mutant from the original program, or *killing* the mutant.

During mutation testing, many mutants are created for the original program. The adequacy of a test set is measured by how many mutants it can kill.

Term1: A mutant is *dead* if a test case has distinguished it from the original program.

Term2: A mutant is *equivalent* if it has the same functional behavior as the original program, and will produce the same results on all inputs.

(We give a more formal definition of equivalence in section 3).

Let K be the number of killed mutants, M be the number of total mutants generated, and E be the number of equivalent mutants. Then the mutation score [DLS78] is computed as:

$$\text{Mutation Score}(P, T) = K/(M \Leftrightarrow E) * 100\%$$

We call a test data set, T, *mutation-adequate* for a program P, if its mutation score is 100% for P, i.e., $K = M \Leftrightarrow E$.

In practice, a tester interacts with an automated mutation system to determine and improve the mutation adequacy of a test data set. This forces the tester to test for specific types of faults. These faults are represented as simple syntactic changes to the test program that create mutant programs.

The mutation system *Mothra* [DGK+88, KO91] uses 22 mutant operators to test Fortran-77 programs. These operators are shown in Table 14 in Appendix C. These 22 mutant operators can be divided into three categories based on the syntactic elements that they modify [OLR+94]. *Operand replacement* operators replace each operand in a program with each other legal operand. Referring to Table 14 in Appendix C, the operators AAR, ACR, ASR, CAR, CNR, CRP, CSR, SAR, SRC, and SVR perform operand replacement. They check for cases where programmers use the wrong variable name or array reference. *Expression modification* operators (ABS, AOR, LCR, ROR, UOI) modify expressions by replacing operators and inserting new operators. They check for cases where programmers make errors inside expressions, for example, using the wrong arithmetic operator or an incorrect comparison operator. *Statement modification* operators (DER, DSA, GLR, RSR, SAN, SDL) modify entire statements. They check for statement coverage, statement necessity, and correct label usage. Mothra's mutant operators are defined by King and Offutt [KO91].

Figure 1 is a small Fortran function Min, and a mutant of Min that changes variable I to J. Figure 2 is a in-line version of the same Fortran function Min with five mutated lines (preceded by the Δ symbol).

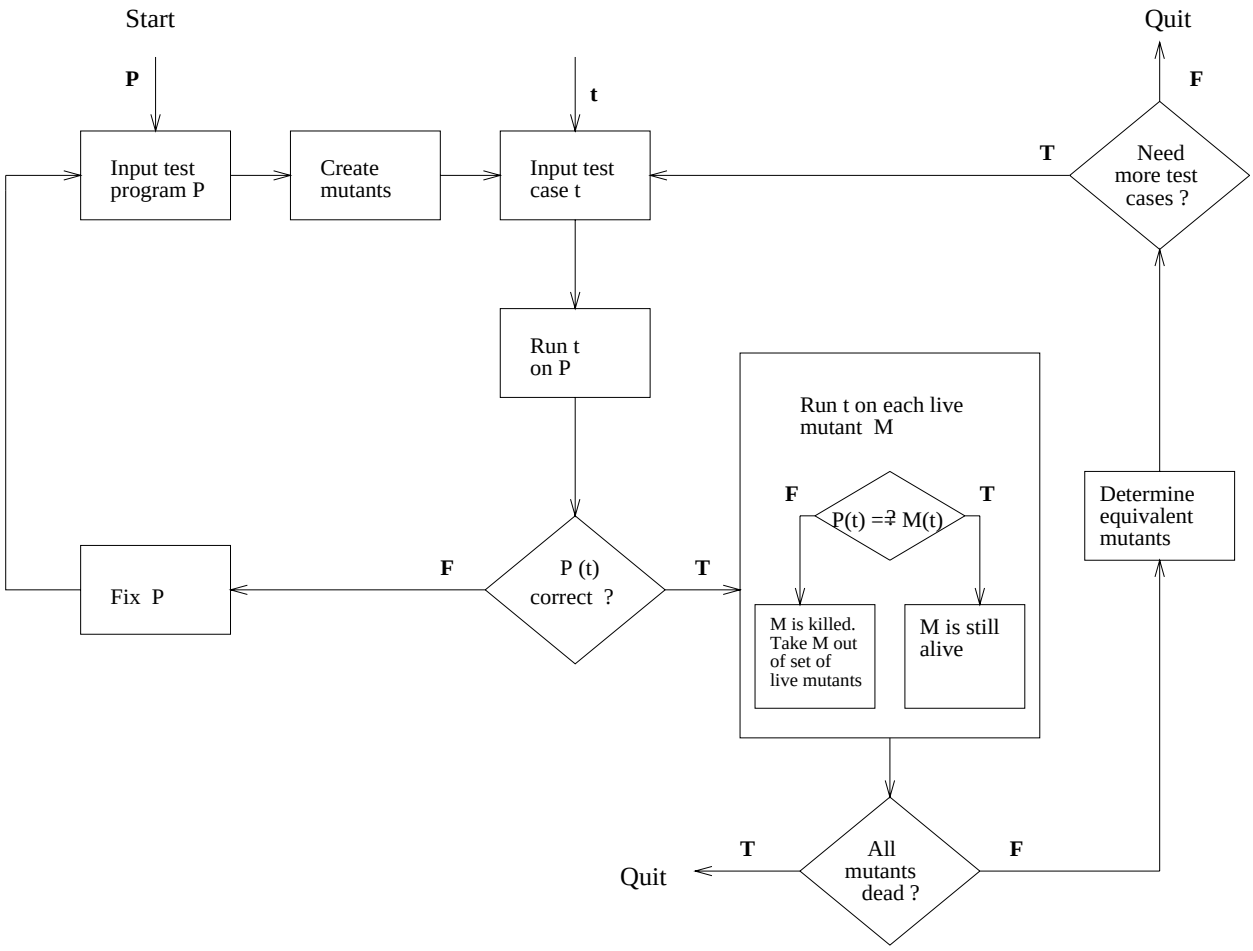

Figure 3: **A Mutation Testing Process**

Note that each mutated statement represents a separate program. The first mutant is the same as shown in Figure 1. The first and fifth mutants in Figure 2 are *operand replacement* mutants, the second and third are *expression modification* mutants, and the fourth is a *statement modification* mutant.
One process that can be used to apply mutation testing is shown in Figure 3. First, a set of mutants is created by an automated tool. Then a tester supplies test data for the original program and checks the output; if the output is not correct, the tester fixes the original and starts again. If it is correct, the same test data is executed on the mutants. If the output from a mutant is different from that of the original, the mutant is considered to be distinguished from the original and is dead. If the output is not different from that of the original, the mutant is said to be still alive. There are two exit points from this process:

- *Exit 1:* All mutants are dead. There is no need to give more test data. The test data that the tester has is adequate.
- *Exit 2:* Some mutants are still alive, but the tester decided not to add more test data.

We do not need to elaborate on exit 1, since the tester has achieved the goal of the mutation testing in this case. But unfortunately, this is rare.
At exit 2, there are some live mutants. Ignoring the financial and management issues, how can a tester decide whether to quit or not? Before the tester can decide to quit the process for technical reasons, the

tester must declare that the mutants left alive are all equivalent and thus there are no test data that can kill them.

For example, consider the mutants in Figure 2. The test cases $(I = \Leftrightarrow 1, J = 3), (I = 2, J = 1)$ are mutation adequate for these mutants. These test cases kill the first through the fourth mutants but not the fifth mutant. Note that in the fifth mutant, the reference to I has been replaced by a reference to MIN. Since these two variables always have the same value at this point in the program, the replacement has no effect on the functional behavior of the program. Thus the output of the mutated program will always be the same as that of the original, and this mutated program is equivalent to the original.

The equivalent mutant in Figure 2 is easy to detect manually. Unfortunately, detecting equivalent mutants is not always easy, and currently must be done by hand. It requires a tester's experience and skills. Thus, it is one of the most expensive parts of the mutation process.

In the next section, we will describe the equivalence problem in detail. A technique for automatically detecting equivalent mutants is presented in section 3; the design and implementation of a tool that uses the technique will be provided in section 4; we give our experimental results of this tool in section 5; and finally, the conclusions and recommendations are in section 6.

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 1: **Equivalent Mutants among Mutant Types** — Programs Used in This Thesis

2 Equivalence Problem

Some of the open problems in mutation testing have been speed of execution, the so called "oracle problem" — that is, determining output correctness, scalability — that is, whether mutation can be applied to big programs, and automatically detecting equivalent mutants. This thesis focuses on the problem of detecting equivalent mutants, which has been a major 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. Meanwhile, detecting equivalent mutants by hand is very time-consuming, which restricts the use of mutation testing.

2.1 Distribution of Equivalent Mutants among Mutant Types

In Budd's dissertation [Bud80], he states that equivalent mutants are not evenly distributed among the 22 mutant types. In fact, the equivalent mutants tend to cluster among only a few types. Table 1 summarizes statistics from the programs used in section 5 of this thesis. The first column in the table gives the mutant operator type and the second column gives the 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.

Table 2 from Offutt and Craft's paper [OC94] and Table 3 from Budd's dissertation [Bud80] show the statistics of distribution of equivalent mutants among mutant types, too. Note that the sets of programs used for the three sets of statistics are different, and Budd's was based on a different mutation system.

These three statistics indicate one very interesting fact. One mutant type, *absolute value insertion* (abs), has many more equivalent mutants than all other mutant types. 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 the value of the expression is unchanged, (this forces the tester to cause each expression to have the value zero, a common testing heuristic).

Mutant Type	% of Equivalent	% of All Mutants
abs	54.3	3.40
scr	16.1	1.70
acr	11.2	0.25
asr	3.9	0.19
svr	3.1	0.18
uoi	3.0	0.15
ror	2.4	0.07
all others	6.0	0.30
total	100.00	6.24

Table 2: **Equivalent Mutants among Mutant Types** — From Offutt and Craft [OC94]

Mutant Type	% of Equivalent	% of All Mutants
abs	75.0	4.0
glr	12.0	0.7
ror	7.5	0.5
all others	5.5	0.5
total	100.00	5.7

Table 3: **Equivalent Mutants among Mutant Types** — From Budd [Bud80]

2.2 Motivation for Automatically Detecting Equivalent Mutants

It is obvious that one motivation for detecting equivalent mutants automatically is that it can save much time and energy for the testers. As said in the previous section, detecting equivalent mutants is one of the most expensive parts of mutation testing. Convincing oneself that a mutant is equivalent is a complicated task that requires an in-depth analysis and understanding of the program.
Another motivation found by Acree [Acr80] is that detecting equivalent mutants automatically also could prevent people from making errors in marking equivalent mutants. Acree defines two types of errors when marking mutants equivalent:

Type 1: Marking a non-equivalent mutant as equivalent.

Type 2: Marking an equivalent mutant as non-equivalent.

Acree chose two programmers to examine 50 mutants in each of four programs. These mutants were chosen randomly from all live mutants after test cases had been developed that eliminated enough mutants so that about half of the remaining mutants were equivalent. The two programmers judged correctly only about 80% of the time, and 12% of the time made type 2 errors and 8% of the time made type 1 errors. Since type 2 errors are "correctable" during later testing, it is really only type 1 errors that require attention. The advantage of using automated techniques to detect equivalent mutants is that an automated tool (if implemented correctly) would avoid type 1 errors since it would not convince itself that a killable mutant was equivalent.

2.3 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, then there is also a computable procedure for generating adequate test data for a program and vice versa. They also show that, in general, neither of these computable procedure 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, we do have one advantage over the general equivalence problem in mutation testing. Specifically, we do not have to determine the equivalence of arbitrary pairs of programs. Because of the definitions of mutant operators, mutants are very much like their original program. Although Budd and Angluin also prove that this problem is undecidable, it has been theorized that for most specific cases, equivalence can be decided [Acr80, OC94]. We can take advantage of this fact to develop techniques and heuristics for detecting many of the equivalent mutants automatically.

2.4 Compiler Optimization Techniques in Equivalence Detection

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

1. Dead code detection,
2. Constant propagation,
3. Invariant propagation,
4. Common subexpression detection,
5. Loop invariant detection, and

6. Hoisting and sinking.

Offutt and Craft [OC94] have designed algorithms for these six techniques, and developed and implemented *Equalizer*, an automated detecting equivalent mutant tool. They found that *Equalizer* detected an average of 10% of the equivalent mutants for 15 programs. They found that the invariant propagation technique is the most powerful technique among these six.

2.5 Using Constraints in Equivalence Detection

In his dissertation [Off88], Offutt describes a technique for using mathematical constraints for testing, which is called Constraint-Based Testing (CBT) [DO91]. He also suggested using CBT to detect equivalent mutants, which is the subject of this thesis. This thesis develops the idea, presents specific strategies and algorithms for detecting equivalent mutants and presents an implementation for these algorithms.

3 Using Constraints To Detect Equivalent Mutants

According to *Webster's Desk Dictionary of the English Language*, a constraint is a confinement or a restriction. A constraint described in this paper 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] is a technique that uses constraints for software testing. It has been used in one implementation of a test data generator and has been shown to be successful [DO91]. Can constraints also be used to detect equivalent mutants? As we described in section 2.3, theoretically and generally speaking, the answer should be "YES". The general approach is to look for infeasibility in constraint systems. In CBT, a constraint represents the conditions under which a mutant will die. That is, if a test case kills the mutant, the constraint system will be true. If the constraint system cannot be true, then there is no test case that can kill the mutant, thus, the mutant is equivalent.

In this section, we give 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, we start our introduction of this technique with constraint-based test data generation. We then give three theorems of how to use CBT to detect equivalent mutants and the proofs for these theorems.

3.1 The Constraint-based Testing Technique

Practical test data generation techniques attempt to choose a subset of the input domain according to some testing criterion. The assumption behind any criterion for generating test data is that the subset of inputs chosen will find a large portion of the faults in the program as well as help the tester establish some confidence in the software. In a mutation system, the tester's goal is to select inputs that cause each mutant to fail. For each mutant, we say that an *effective* test case causes the mutant to fail, and an *ineffective* test case does not. One way to automatically generate test cases to kill mutants is to "filter out" the ineffective test cases. Such a filter may be described by mathematical constraints.

Constraint-based testing 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 a test case t to kill a mutant M that modifies line S of a program P, t has to have three broad characteristics: reachability, necessity and sufficiency.

- **Reachability**

  Since a mutant is represented as a syntactic change to a particular statement, and the other statements in the mutated program are exactly the same as in the original program, it is apparent that as a minimum requirement we must execute the mutated statement. We call this characteristic the *reachability condition.*

  Given a program P and a mutant program M that is formed by changing a statement S in P, if a test case t cannot reach the statement S, it is guaranteed that t will not kill M [DO91].

- **Necessity**

  A program that generates test cases to kill mutants needs to have one simple, broad characteristic: it must generate test data that makes a difference in the mutant's behavior. We call this characteristic the *necessity condition.*

Given a program P and a mutant program M that is formed by changing a statement S in P, for a test case t to kill M, it is *necessary* that if S is reached, the state of M immediately following some execution of S be different from the state of P at the same point [DO91].

To see why, simply note that since M is syntactically equal to P except for the mutated statement S, if the states of the two programs do not differ after the last execution of S, they will never differ.

- **Sufficiency**

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

Let D be the domain of all test cases t for P. This domain is represented by all t to P and the range of values that t can assume. Each t in D represents a possible test case for P. In the light of the 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, we derive the following facts:

Fact 1: t is an effective test case that will kill a mutant $M \Leftrightarrow t \in D_s$

Fact 2: t is an effective test case that will kill a mutant $M \Rightarrow t \in D_r \cap D_n$

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

Unfortunately, finding t such that $t \in D_r$ is undecidable [Off88]. This is because the reachability condition requires that if t satisfies the condition then the statement S will be executed, which requires a solution to the halting problem. A weaker condition is that if S is executed, then the condition will be true. We use C_R to refer to this weaker condition and D_R to refer to a domain that contains all inputs t that satisfy C_R. Since $C_r \Rightarrow C_R$, it is clear that:

Fact 4: $D_r \subseteq D_R$.

Figure 4 is a Venn diagram that graphically shows these domains and their relationships for one mutant. We will use the above facts and this diagram in later proofs.
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.
Since mutant 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*, which is an algebraic expression, to describe this necessity condition C_n, that is, if a test case that satisfies the necessity condition will be executed on the mutated statement, the state immediately following some execution of the mutated statement will be incorrect.
As an example, Figure 5 shows the function *Mid* with a mutant that replaces the relational operator *LT* with *LE* on statement 5. The path expression for statement 5 is $(Y < Z) \wedge (X \geq Y)$, and the necessity constraint for the mutant is $((X < Z) \neq (X \leq Z))$.
In CBT, a test case t is generated 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. It is clear that t is in the union of D_R and D_n, i.e., $t \in D_R \cup D_n$.

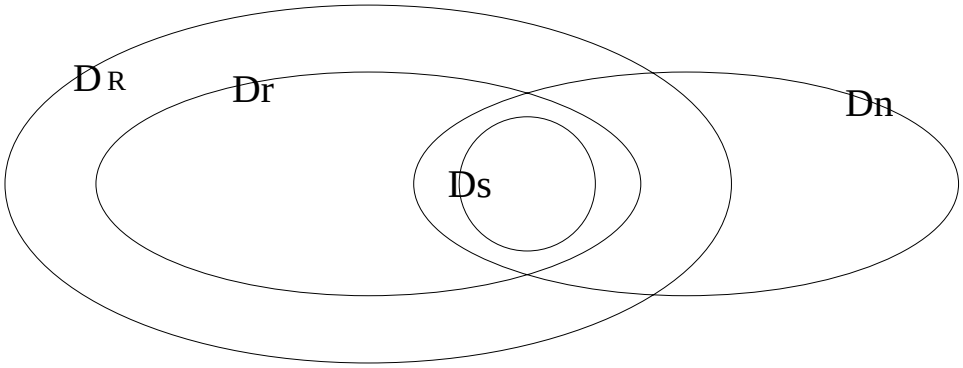

Figure 4: **Input Domain Subsets**

```
         FUNCTION Min (X, Y, Z)
         INTEGER X, Y, Z
1        Mid = Z
2        IF (Y.LT.Z) THEN
3           IF (X.LT.Y) THEN
4             MID = Y
5           ELSE IF (X.LT.Z) THEN
     Δ      ELSE IF (X .LE. Z) THEN
6             MID = X
7           ENDIF
8        ELSE
9           IF (X.GT.Y) THEN
10            MID = Y
11          ELSE IF (X.GT.Z) THEN
12            MID = X
13          ENDIF
14       ENDIF
15       RETURN
16       END
```

Figure 5: **Function Mid**

3.2 Why Can Constraints be Used to Detect Equivalent Mutants?

In section 1.2, we said that an equivalent mutant has the same functional behavior as the original program. However, the term "the same functional behavior" was not formally given. Now we formally define equivalent mutants in terms of inputs and outputs.

Definition: Let P be a program, and M a mutated program of P. 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, then we will not be able 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 our efforts in automatically detecting equivalent mutants, we state following three theorems, and give proofs for these theorems. These proofs are based on the definition of an equivalent mutant, descriptions of input domains and facts given in 3.1.

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, that is, 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, that is, 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, that is, $D_r \cap D_n$ 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 we use the weaker reachability condition (C_R) instead of the reachability condition (C_r), since $C_r \Rightarrow C_R$, we can easily have following two derivations:

- $D_R = \emptyset \Rightarrow M$ 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

In section 3.1, we mentioned that CBT uses *path expression constraint systems* to represent reachability conditions (the weaker conditions) and *necessity constraints systems* to represent necessity conditions. So 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 greater than 0 and less than 0 at the same time. If a mutant M has the above constraint as a path expression associated with it, then we can say M is equivalent.
So far, we have translated the problem of detecting equivalent mutants to the problem of recognizing a contradiction in a constraint system, which is a mathematical expression. Test data generation uses constraints by satisfying constraints to generate test cases, while equivalent mutant detection uses constraints by recognizing infeasible constraints to detect equivalent mutants. Figure 6 shows these two uses of constraints.

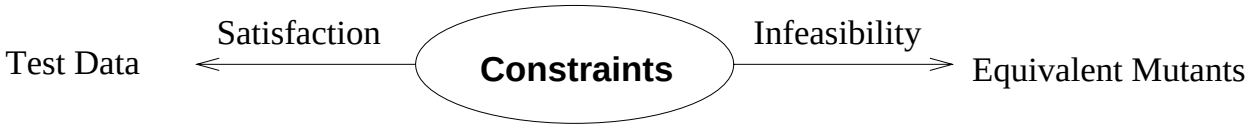

Figure 6: **Using Constraints**

3.3 Constraint Representation

Before we describe 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, and a *disjunctive clause* uses only the logical OR. We keep all constraints 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 clause.
In CBT, a *constraint system* is referred to as a DNF clause. DNF is used for convenience during constraint generation (each conjunctive clause within a path expression represents a unique path to a statement).

3.4 Strategies for Detecting Equivalent Mutants

As mentioned in the previous section, we try to find a contradiction in the constraint system to detect equivalent mutants. Unfortunately, recognizing whether an arbitrary constraint system is infeasible is an undecidable problem [DO91]. Thus, we cannot always recognize whether a constraint system is infeasible. Presently, however, equivalent mutants are detected by hand, so even partial solutions are worthwhile. We apply a collection of special case analysis and heuristics to recognize infeasible constraint systems, and equivalent mutants, when possible. In section 5 we present empirical results, from an implementation, that measures how well these strategies work.
In this thesis, we use three broad strategies that attempt to recognize infeasible constraint systems. They are negation, constraint splitting and constants comparison.

3.4.1 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 non-overlapping but not domain covering. We introduce partial negation 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 constraints that are semantically equal if they are syntactically equal.
Examples:

- Let A be the constraint $x > 1$ and B be the constraint $x \leq 1$. Then A is the *negation* of B, or B is the *negation* of A (negation is commutative). Both constraints cannot be satisfied at the same time, 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 the *partial negation* of B, or B is the *partial negation* of A. Both constraints cannot be satisfied at the same time, but the domain of x that makes the two constraints TRUE do 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 a variable over integers, A be the constraint $x > 0$ and B be the constraint $x \geq 1$. Thus A and B are not syntactically equal, but they are semantically equal.

The negation strategy is the basic way we recognize infeasible constraints. Given two constraints, we first use *negation* or *partial negation* to rewrite one of the constraints, then compare these two constraints. 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'. Then, since A' and B are syntactically equal, A and B conflict.

3.4.2 Constraint splitting

We also use *constraint splitting* to recognize infeasible constraints. The motivation is as follows. We found that 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 tell that the necessity constraint conflicts with the path expression.
To detect such conflicts, we use a strategy called *constraint splitting*. Given two constraints (say C and D), we generate two new constraints (say A and B), such that $C \Rightarrow A \vee B$. Then we compare A and B to D. We prove that if both A and B conflict with D, then C conflicts with D, and thus show the correctness of the "constraint splitting" strategy.

	$C \Rightarrow A \vee B$	
$\Leftrightarrow$	$\neg C \vee (A \vee B)$	— implication
$\Leftrightarrow$	$(A \vee B) \vee \neg C$	— commutativity
$\Leftrightarrow$	$\neg\neg(A \vee B) \vee \neg C$	— logical negation
$\Leftrightarrow$	$\neg(\neg A \wedge \neg B) \vee \neg C$	— De Morgan's law
$\Leftrightarrow$	$\neg A \wedge \neg B \Rightarrow \neg C$	— implication

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, we analyze the cases shown in Table 4. 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.

3.4.3 Constant comparison

A third strategy to decide whether two constraints conflict is based on a property that is common in the constraints generated for test cases. The property is that both constraints should have the format (V rop K), where V is a variable, rop is a relational operator, and K is a constant. Also the variables in both constraints are required to be the same.
Let A be the constraint (X rop1 K1), and B be the constraint (X rop2 K2). By evaluating two constants and two relational operators, we decide whether A conflicts with B. We call this strategy *constants comparison*. Table 5 shows the cases we 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. Note that it does not always have such a predicate available. The last column is the conclusion of whether the two constrains 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; and the letter "F" stands for "False" which means the two given constraints do not 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**

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
$x \neq k1$	$x \neg k2$	—	F

Table 5: **Constant Comparison Cases Analysis**

The reason to use this strategy is that the negation strategy described before is based on syntactic checking of two constraints. For example, given the two constraints $(x > 1)$ and $(x < 0)$, if we use the negation strategy, we can partially negate or negate the first constraint $(x > 1)$ 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 they conflict. But if we use "constants comparison", we can tell that they do conflict.
To expand the use of constant comparison, if a constraint has the format (V aop K1) rop K2, we rewrite it 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.

4 Design And Implementation

We have implemented a tool *Equivalencer* that uses CBT to detect equivalent mutants. Equivalencer is integrated with *Godzilla*, a test data generator in *Mothra*. Although the technique (CBT) is language independent, the tool detects equivalent mutants for Fortran 77 programs. *Equivalencer* was implemented in the programming language *C* on a Sun Sparc classic workstation running the SunOS 4.1.3 (MULTICAST-4.1.3) #2 operating system. *Equivalencer* contains more than 2000 lines source code and also uses several *Mothra* and *Godzilla* library functions.
In this section, we first describe the assertion constraints that affect our design and implementation; and how we insert them into a program under test. Then we give the architectural design for the tool *Equivalencer*. Finally, the algorithms that implement the strategies described in section 3 are given.

4.1 Inserting Assertion Constraints

In our implementation, we use assertion constraints to help detect equivalent mutants. Assertions are constraints that a user inserts into a program under test to manually restrict the input domain of some variables. They could be preconditions to the program or predicates on a specific statement, 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 (semantically). These are often part of the specifications, that is, preconditions. The assertions on the internal variables can sometimes be derived automatically (syntactically). We have three kinds of assertions in the programs used in this thesis.

1 . Assertions on parameter variables, such as (F.GE.1 .AND. F.LE.N .AND. N.GE.1 .AND. N.LE.10) in the function *Find* shown in Figure 7. Assertions like these are also known as preconditions.

2 . Assertions on internal variables that could be derived automatically, such as (I.GE.1 .AND. M.GE.1) in the function *Find* shown in Figure 7. There are known techniques for doing this, such as slicing [Wei84] and control flow analysis [FL88]. The purpose of implementing a tool in this thesis is to show how we can use constraints to detect equivalent mutants, so in our implementation, we generate these assertions by hand and put them in as assertion constraints.

3 . Assertions on internal variables that could not easily be derived automatically, such as (J.GE.0 .AND. NS.GE.1) in the function *Find* shown in Figure 7. Although creating assertions is an additional burden on the tester, we feel that it is easier for the tester to provide information about variables in the program than to analyze mutants. We can imagine a system that interacts with the tester and provides information to help the tester to analyze mutants and to detect equivalent mutants on a program under test.

When *Godzilla* generates a constraint that has arrays in it, it takes a safe 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 we describe below.
We use a constraint with arrays to recognize contradiction only if this constraint is an assertion constraint. That is because we can be sure that an array in an assertion constraint that we insert will apply to every element of 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 $A() < 0$, we can be sure that this assertion conflicts with this necessity constraint. To take advantage of this, we added a routine that checks whether assertion constraints conflict with other constraints, such as necessity, path expression constraints. In this routine, *Equivalencer* works on array constraints. We will refer to this routine as *array-extension*.

4.2 Architectural Design

Figure 8 is the overall architectural diagram of *Godzilla* after adding *Equivalencer*. The tools are represented in the figure by boxes. These tools, which are designed as separate entities and implemented as separate

```
      SUBROUTINE FIND (A, N, F)
      INTEGER A (10), N, F

C= A  inout
C= N  in
C= F  in

C     F is index into A(). After execution, all elements to the left of
C     A(F) are less than or equal to A(F) and all elements to the right
C     of A(F) are greater than or equal to A(F).
C     Only the first N elements are considered.
C     From DeMillo, Lipton, and Sayward [DeMi78], repeated from Hoare's
C     paper [Hoar70].

      INTEGER M, NS, R, I, J, W

C     External parameters assertions:
      ASSERT (F.GE.1.AND.F.LE.N.AND.N.GE.1.AND.N.LE.10)
C     Internal variables assertions:
      ASSERT (I.GE.1.AND.M.GE.1)
C     Internal variables assertions (hard to derive automatically):
      ASSERT (J.GE.0 .AND. NS.GE.1)
      M = 1
      NS = N
10    IF (M.GE.NS) GOTO 1000
      R = A (F)
      I = M
      J = NS
20    IF (I.GT.J) GOTO 60
30    IF (A(I).GE.R) GOTO 40
      I = I + 1
      GOTO 30
40    IF (R.GE.A(J)) GOTO 50
      J = J - 1
      GOTO 40
50    IF (I.GT.J) GOTO 20
      W = A (I)
      A (I) = A (J)
      A (J) = W
      I = I + 1
      J = J - 1
      GOTO 20
60    IF (F.GT.J) GOTO 70
      NS = J
      GOTO 10
70    IF (I.GT.F) GOTO 1000
      M = I
      GOTO 10
1000  RETURN
      END
```

Figure 7: **Subroutine Find**

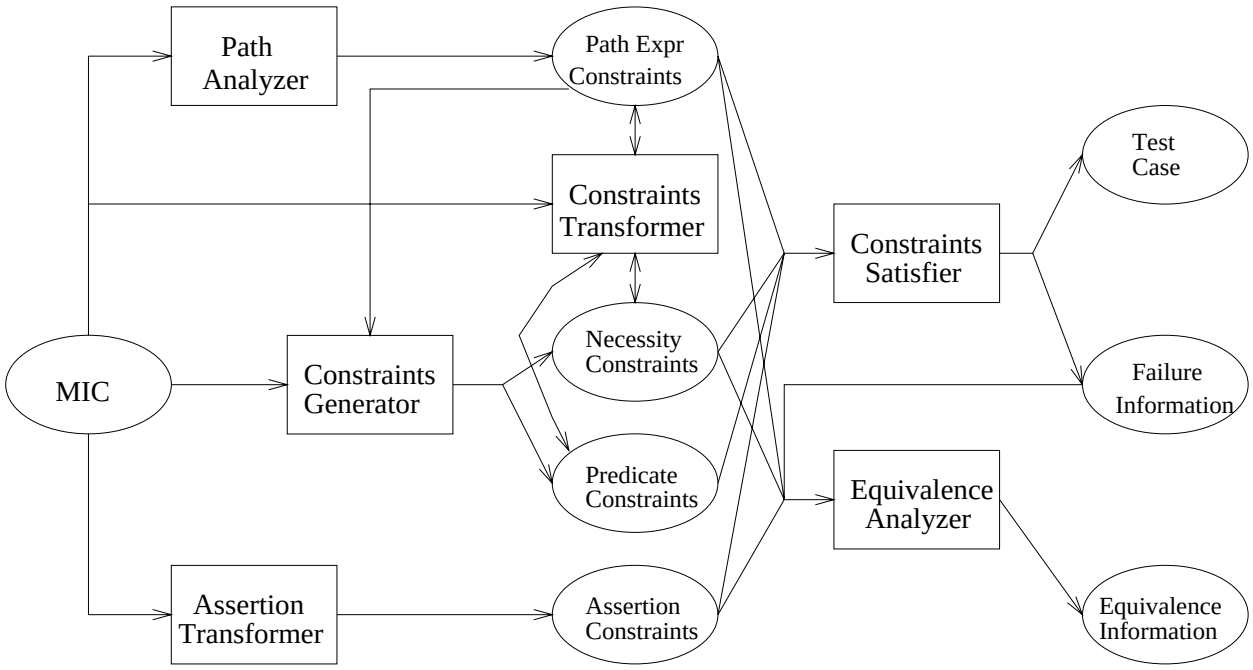

Figure 8: **Godzilla Architectural Diagram**

programs, communicate through files (data) represented by ovals. The arrows indicate the flow of data through *Godzilla*.
The *MIC* (Mothra Intermediate Code) is generated by the *Parser* tool in *Mothra* (not shown in the diagram), from parsing a Fortran test program. The Path Analyzer, *Paths*, uses the *MIC* to generate *Path Expression Constraints*. The Constraints Generator, *Congen*, uses *MIC* to generate *Necessity Constraints*; it also uses the *Path Expression* to generate the *Predicate Constraints*. The Assertion Transformation (*Transass*) uses *MIC* to generate *Assertion Constraints* (if they exist). The Constraints Satisfier (*Consat*) generates test data by satisfying the constraints; and it also provides *Failure Information* (why a test case cannot be generated). Then *Equivalencer* (Equivalence Analyzer) uses the *Path Expression, Necessity*, and *Assertion Constraints* that are generated by the other tools to detect equivalent mutants by using the strategies described in section 3. Also, *Equivalencer* uses the *Failure Information* offered by *Consat* to detect some equivalent mutants. We could not find a way to use *Predicate Constraints*.
Figure 9 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 *Consat*. 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) 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 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 exits.

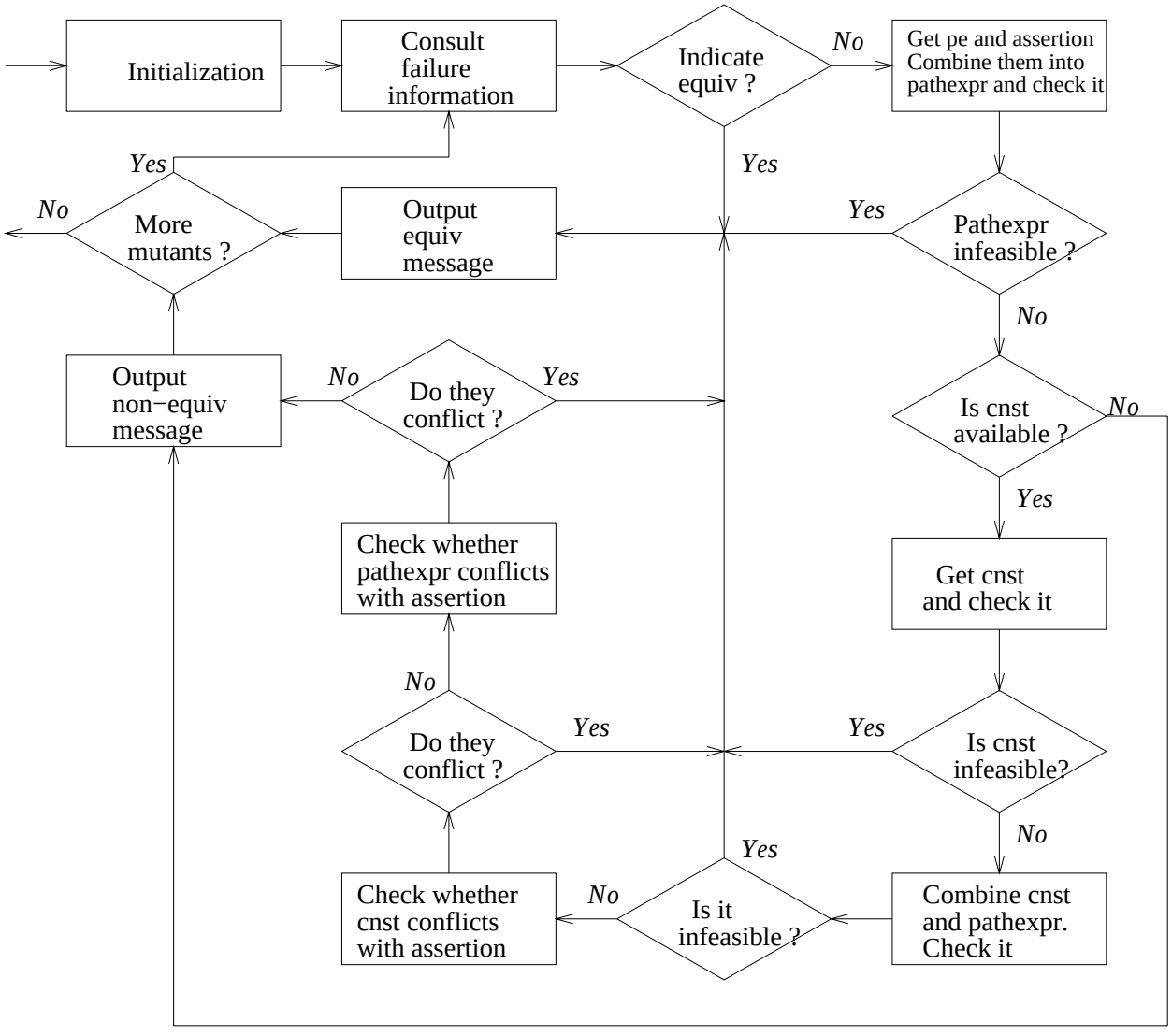

Figure 9: **Equivalencer Execution Diagram**

4.3 Algorithms

Following are the algorithms for the three strategies described in section 3. They are Negation, Constraint Splitting, and Constants Comparison.

Negation algorithm

```
input       two constraints A and B
output      boolean  — If A and B conflict, return conflict, else return no-conflict.
declare     neg-A, partneg1-A, partneg2-A  — negated or partial negated constraint
BEGIN
   neg-A = Negate(A)            — according to Table 6 (negation table)
   IF neg-A syntactically equals B
      RETURN conflict
   ELSE
      IF the relation operator in A is one of {>, <, =}
         partneg-A = PartialNegate1(A)          — according to Table 6
         IF partneg1-A syntactically equals B
            RETURN conflict
         ELSE
            partneg2-A = PartialNegate2(A)          — according to Table 6
            IF partneg2-A syntactically equals B
               RETURN conflict
            ELSE
               RETURN no-conflict
            END IF
         END IF
      END IF
   END IF
END Negation
```

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 6: **Negation and Partial Negation**

Constraint splitting algorithm

```
input       necessity constraint C, and path expression D
output      boolean   — If C and D conflict, return conflict, else return no-conflict
declare     A, B      — constraints
BEGIN
   IF the format of C is not ((V1[1]  aop V2[2]) rop[3]  K[4])
      RETURN no-conflict
   ELSE          — according to the Table 4 (constraint splitting table)
      A = NewConstraint1(C)
      B = NewConstraint2(C)
   END IF
   use negation strategy to check for a conflict          — see negation algorithm
   IF A and B both conflict with D
      RETURN conflict
   ELSE
      use constants comparison strategy to check for a conflict
                — see constants comparison algorithm
      IF A and B both conflict with D
         RETURN conflict
      ELSE
         RETURN no-conflict
      END IF
   END IF
END Constraint Splitting
```

[1] V1 is a variable
[2] V2 is a variable
[3] rop is a relational operator
[4] K is a constant

Constants Comparison Algorithm

input two constraints A and B
output boolean — If A and B conflict, return conflict, else return no-conflict.

BEGIN
  IF the format of A is (V[5] rop[6] K[7])
    keep the format
  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[8] K1[9]) rop K2[10])
      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
  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
  IF (ConstantsComparison(A, B) == True)
    — according to Table 5 (constants comparison table)
    RETURN conflict
  ELSE RETURN no-conflict
  END IF
END Constants Comparison

[5] V is a variable
[6] rop is a relational operator
[7] K is a constant
[8] aop is one of the arithmetic operators ($+, -, \times, \div$)
[9] K1 is a constant
[10] K2 is a constant

Program	Description	Statements	Mutants	Equivalent
Bsearch	Binary search on an integer array	20	299	27
Bub	Bubble sort on an integer array	11	338	35
Cal	Days between two dates	29	3010	236
Euclid	Greatest common divisor (Euclid's)	11	196	24
Find	Partitions an array	28	1022	75
Insert	Insertion sort on an integer array	14	460	46
Mid	Median of three integers	16	183	13
Pat	Pattern matching	17	513	61
Quad	Real roots of quadratic equation	10	359	31
Trityp	Classifies triangle types	28	951	109
Warshall	Transitive closure of a matrix	11	305	35

Table 7: **Experimental Programs**

5 Results

A principal reason for implementing *Equivalencer* was to provide a vehicle for detecting equivalent mutants automatically to improve the mutation testing process. Therefore, a number of experiments have been performed on several programs to determine the equivalent mutant detection power of *Equivalencer*. Results for different combinations of the strategies and combined with different tools in the implementation are presented. A distribution of detected mutants by type is given. The time to detect equivalent mutants of these programs is also presented. The results of a comparison with compiler optimization is presented, too. A test suite of eleven Fortran-77 programs was chosen 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 thesis does not attempt to solve that problem. However, the eleven 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. The eleven programs are *Bsearch, Bub, Cal, Euclid, Find, Insert, Mid, Pat, Quad, Trityp* and *Warshall*. These programs have been studied by several researchers and include a very carefully created list of equivalent mutants under *Mothra*.

Table 7 lists these programs along with a brief description, the number of executable Fortran lines, the number of mutants generated, and the number of equivalent mutants. The source code for each program is shown in Appendix A. These programs are small for two reasons. First, mutation analysis is primarily intended for unit-level (subroutines and functions) testing. Second, this study involved a lot of hand analysis; for each program and each mutated program studied, equivalent mutants had to be identified.

In *Godzilla*, every variable in each constraint has a statement number associated with it. 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, we might have the same statement number for the variable. 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. The tools *Transass* and *Conform* in *Godzilla* are affected by these statement numbers. We describe them below.

There are two options to *Transass*. Option [-p] propagates assertions through the program so that the assertions apply to all statements. But it assumes that the variables will not change values, or if the values have been changed, they will not affect the validity of assertion constraints. So *Transass* with this option assigns the same statement number, which is 1, to the assertion constraint on each statement. Option [-sp] propagates assertions through the program, and assigns the corresponding statement line number to

every statement. If no option is given, *Transass* assumes that the assertion only applies to the statement immediately following the assertion.
Another *Godzilla* tool, *Conform*, has greatly effected our analysis. *Conform* re-writes constraints to be solely in terms of the input parameters by symbolically evaluating the program [Off91]. Also, *Conform* re-writes the statement numbers in the constraints. Initially, the statement number each variable is the statement where the constraint appears. *Conform* re-writes a variable's statement numbers to a previous statement number if the variable has not been assigned a new value from the previous statement. We refer to this functionality of *Conform* as *constraints propagation* because the statement numbers are "propagated" through the constraints. By getting help from *Conform* to propagate constraints, especially on path expression constraints, *Equivalencer* increases its detection ability. Unfortunately, *Conform* has a design flaw that causes it to work incorrectly on constraints involving loops. If a constraint re-written by *Conform* is in a loop, it is correct for the first iteration, but usually not correct in the second and subsequent iterations. Since *Equivalencer* detects equivalent mutants using constraints, *Equivalencer* will detect incorrect equivalent mutants if the constraints are wrong. Unfortunately, fixing the design flaw for *Conform* in the limited time available was impractical. But in order to show that *Conform* can help *Equivalencer*, we present an experiment with *Conform* in which we removed the incorrect equivalent mutants by hand, leaving the correct equivalent mutants.
For each experiment, the first few steps are the same. All mutants were generated by a tool called *Mutmake* in *Mothra*, the path expression constraint for each statement were generated by a tool called *Paths* in *Mothra*, and the necessity constraint for each mutant (if it could be generated) were generated by *Congen*. Then different steps or options were used for each experiment. We describe the rest of each experiment as follows:

Experiment 1: *Transass* was run with option [-sp] to generate assertion constraints (if there were any), then *Equivalencer* with only the negation strategy was run on every mutant to detect whether it is equivalent.

Experiment 2: *Transass* was run with option [-sp], then *Equivalencer* with the negation and constraint splitting strategies was run on every mutant to detect whether it is equivalent.

Experiment 3: *Transass* was run with option [-sp], then *Equivalencer* with constant comparison strategy was run on every mutant to detect whether it is equivalent.

Experiment 4: *Transass* was run with option [-sp], then *Equivalencer* was run on every mutant to detect whether it is equivalent by using all three strategies.

Experiment 5: *Transass* was run with option [-sp], then *Equivalencer* was run on every mutant to detect whether it is equivalent by using all three strategies plus array extension. For this experiment, we captured the starting time from Unix command "date" just before executing *Equivalencer*, and captured the ending time from date just after the execution finished. We measured the time for this experiment because it used all three strategies plus array extension.

Experiment 6: This time, *Transass* was run with option [-p] to generate assertion constraints (if there were any), then *Conform* was run to re-write the constraints; and then *Equivalencer* was run on every mutant to detect whether it is equivalent by using all three strategies plus array extension.

Table 8 shows the results from experiments 1 through 5. The first column in the table gives the names of program used in experiments. The second column gives the number of equivalent mutants in each program (as determined by hand analysis). The third through the last columns give the percentage of equivalent mutants that were detected by using the different combinations just described. S123a is from experiment 5, S1 is from experiment 1, S12 is from experiment 2, S3 is from experiment 3, and S123 is from experiment 4. From Table 8, we can see that the average detection percentage is about 30% by using all the strategies plus array-extension. Among the three strategies (negation, constraint-splitting and constants-comparison), we found that the constants-comparison strategy is the most powerful.
Table 9 shows that *Conform* could help *Equivalencer*, the results from experiment 5 and 6. As in Table 8, the first column in the table gives the names of the program used in the experiments and the second column

Program	Equiv. num.	S123a %	S1 %	S12 %	S3 %	S123 %
Bsearch	27	25.93	22.22	25.93	14.81	25.93
Bub	35	65.71	45.71	45.71	65.71	65.71
Cal	239	12.55	0.84	0.84	11.72	11.72
Euclid	24	70.83	62.50	70.83	50.00	70.83
Find	75	62.67	12.00	12.00	50.67	62.67
Insert	46	50.00	15.22	15.22	36.96	50.00
Mid	13	0.00	0.00	0.00	0.00	0.00
Pat	61	44.26	1.64	1.64	36.07	44.26
Quad	31	9.68	0.00	0.00	9.68	9.68
Trityp	109	11.93	6.42	6.42	6.42	11.93
Warshall	35	54.29	0.00	0.00	45.71	45.71
Total/Avg.	695	30.07	9.06	9.50	24.46	29.35

Table 8: **Detection Percentage by Using Different Strategies**

gives the number of equivalent mutants in each program. The third and fourth columns give the number and percentage of detected equivalent mutants without using *Conform*. The fifth and sixth columns are the number and percentage of detected equivalent mutants using *Conform*. Since the sets of detected equivalent mutants with and without using *Conform* intersect, the seventh and eighth columns give the number and percentage of the union of these two sets. The bugs in *Conform* cause *Equivalencer* to erroneously mark killable mutants as equivalent mutants. Since the bugs in *Conform* are beyond our control, we removed the incorrectly detected mutants by hand.

Table 10 shows the time for detecting equivalent mutants on these programs from experiment 5. The first column is the program's names, and the second and third columns are the number of mutants generated for the program and the number of equivalent mutants. The fourth column is the number of detected equivalent mutants. The last column is the wall-clock execution time of *Equivalencer* on each program in seconds.

From Table 10, we can see the longest time to detect one equivalent mutant is less than 6 seconds in the *Cal* program. In most programs, the time to detect one equivalent mutant is less than 1 second. We did not try to optimize these programs, thus, it is likely that a commercial tool could be quite a bit faster.

Table 11 shows the distribution of detected equivalent mutants among the mutant types (based on the union column of Table 9). The first column is the mutant type and the second column is the number of equivalent mutants. The third column gives the number of detected equivalent mutants by each type, the fourth gives the percentage of detected equivalent mutants by each type of equivalent mutants, and the last gives the percentage of detected equivalent mutants on all equivalent mutants. We can see that most of the detected equivalent mutants belong to mutant type *ABS* and above 90% have been detected. Since necessity constraints are not built for every mutant type, only 14 of the 22 mutant types have necessity constraint templates; the necessity constraint templates are shown Appendix B. Among the programs used in our experiments, 3 mutant types do not have equivalent mutants.

To compare with compiler optimization techniques [OC94], we chose the same programs that were used in Offutt and Craft's paper [OC94] and in our experiments. They are *Bsearch, Bub, Cal, Euclid, Find, Insert, Mid, Trityp* and *Warshall*. The results are shown in Table 12. The first column is the program names and the second column is the number of equivalent mutants. The third and fourth columns are the number of detected equivalent mutants and the percentage of detection by using compiler optimization detection techniques (Optimization) respectively. The fifth and sixth columns are the number of detected equivalent mutants and the percentage of detection by using the constraints detection technique (Constraints) respectively. The data of using compiler optimization techniques is from Offutt and Craft's paper [OC94]. The data on constraints detection is from the union column in Table 9.

Program	Equiv. num.	without Conform num.	without Conform %	with Conform num.	with Conform %	union num.	union %
bsearch	27	7	25.93	19	70.37	19	70.37
bub	35	23	65.71	4	11.43	24	68.57
cal	239	30	12.55	33	13.81	37	15.48
euclid	24	17	70.83	16	66.67	18	75.00
find	75	47	62.67	63	84.00	63	84.00
insert	46	23	50.00	31	67.39	32	69.57
mid	13	0	0.00	3	23.08	3	23.08
pat	61	27	44.26	21	34.43	29	47.54
quad	31	3	9.68	4	12.90	4	12.90
trityp	109	13	11.93	80	73.39	80	73.39
warshall	35	19	54.29	3	8.57	22	62.86
total/avg	695	209	30.07	277	39.86	331	47.63

Table 9: **Detection Results of with/without Using Conform**

Program	Mutants	Equivalent	Detected	Time (s)
Bsearch	299	27	7	4
Bub	338	35	23	10
Cal	3010	236	30	167
Euclid	196	24	17	3
Find	1022	75	47	38
Insert	460	46	23	5
Mid	183	13	0	0
Pat	513	61	27	17
Quad	359	31	3	1
Trityp	951	109	13	27
Warshall	305	35	19	10
Total	7636	695	209	281

Table 10: **Detecting Time**

Mut Type	Equiv Mut	Detected	% on Each Type	% on All Types
abs	328	303	92.38	43.60
aor	2	1	50.00	0.14
der	6	3	50.00	0.43
csr	7	3	42.86	0.43
svr	31	6	19.35	0.86
ror	25	4	16.00	0.58
sar	7	1	14.29	0.14
scr	49	7	14.29	1.01
asr	8	1	12.50	0.14
rsr	10	1	10.00	0.14
uoi	42	1	2.38	0.14
all others	180	0	0.00	0.00
total/avg.	695	331	47.63	47.63

Table 11: **Detected Equivalent Mutants among Mutant Types**

	Equiv.	Optimization		Constraints	
Program	num.	num.	%	num.	%
Bsearch	27	0	0.00	19	70.37
Bub	35	5	14.29	24	68.59
Cal	239	0	0.00	37	15.48
Euclid	24	1	4.17	18	75.00
Find	75	1	1.33	63	84.00
Insert	46	10	21.74	32	69.57
Mid	13	1	7.69	3	23.08
Trityp	109	12	11.01	80	73.39
Warshall	35	4	11.43	22	62.86
Total/Avg.	603	30	4.98	298	49.42

Table 12: **Comparison on Compiler Optimization and Constraints Detections**

From Table 12, we see that the constraint-based technique is more powerful than the compiler optimization techniques in detecting equivalent mutants. The average percentage of equivalent mutants detected by the constraint-based technique is almost 10 times more than the average percentage detected by the compiler optimization techniques.

6 Conclusions And Recommendations

In this section, we give the conclusions first and then present our recommendations. The recommendations are both on improving the detection techniques and on improving the software.

6.1 Conclusions

In this thesis, we have presented a partial solution to the problem of equivalent mutant detection. The solution was proposed, specific algorithms were developed and a proof of concept experimental tool was built. Our results show that our approach is an effective solution to this problem.
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 9 in section 5). With appropriate extensions (see section 6.2), we think the detection percentages could be even higher. Comparing with the *Equalizer* [OC94], which uses the compiler optimization techniques, *Equivalencer* is much more powerful at detecting equivalent mutants. Also the detecting time is reasonably fast, even in our implementation, which was optimized for speed. Considering that every test case would have to be run against those equivalent mutants, the time saving is large. Considering the time to detect those equivalent mutants by hand, the time saving is significant.
Since *Equivalencer* uses constraints to detect equivalent mutants, it needs other tools to generate and modify constraints. This technique should use the following process to get the best results.

1. Run *Parser* to parse a program under test
2. Run *Mapper* to initialize parameter variable's class
3. Run *Mutmake* to generate mutants for the program
4. Run *Transass* with the option [-sp] to generate assertion constraints
5. Run *Paths* to generate path expression constraints
6. Run *Congen* to generate necessity constraints
7. Run *Consat* to generate test cases. We need to run *consat* because we need a side product from it; the failure information, which gives information about equivalent mutants
8. Run *Equivalencer* to detect equivalent mutants
9. Run *Transass* with the option [-p] to generate assertion constraints
10. Run *Conform* to modify constraints
11. Run *Equivalencer* to detect equivalent mutants

6.2 Recommendations

In this subsection, we make several recommendations for improving the equivalent mutant detection power. We divide these into two categories: recommendations for improving the technique, and recommendations for improving the software.

6.2.1 Improving the detection techniques

We recommend three ways to improve our techniques. One is a 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.

More infeasible constraint recognition strategies

Another strategy we found that could detect equivalent mutants by recognizing infeasible constraints is the following. Assume we have a constraint X. Suppose we can find some other constraints, say $C1, C2, ..., Cn$, such that $X \wedge C1 \wedge C2 \wedge ... \wedge Cn \Rightarrow Y$, and $C1, C2, ..., Cn$ are true. If we can prove that $\neg Y$ holds, then we can say $\neg X$ holds, which means that X is an infeasible constraint. This strategy is proved below:
Given: $C1, C2, ..., Cn$, and $(C1 \wedge C2 \wedge ... \wedge Cn \wedge X \Rightarrow Y)$,

$$
\begin{array}{ll}
C1 \wedge C2 \wedge ... \wedge Cn \wedge X \Rightarrow Y & \text{— premise} \\
\Leftrightarrow \neg(C1 \wedge C2 \wedge ... \wedge Cn \wedge X) \vee Y & \text{— implication} \\
\Leftrightarrow Y \vee \neg(C1 \wedge C2 \wedge ... \wedge Cn \wedge X) & \text{— commutativity} \\
\Leftrightarrow \neg Y \Rightarrow \neg(C1 \wedge C2 \wedge ... \wedge Cn \wedge X) & \text{— implication} \\
\Leftrightarrow \neg Y \Rightarrow \neg(True \wedge True \wedge ... \wedge True \wedge X) & \text{— premise} \\
\Leftrightarrow \neg Y \Rightarrow \neg X & \text{— And simplification}
\end{array}
$$

One specific case we analyzed is as follows. Constraint X is $((x+y) < 0)$, and constraints $(x > 0), (y < 0)$ are true. We can have $((x + y) < 0) \wedge (x > 0) \wedge (y < 0) \Rightarrow (abs(y) > x)$ and if we can show that $\neg(abs(y) > x)$ holds, then we can say $\neg((x + y) < 0)$ holds.
Another specific case we analyzed is the following. Constraint X is $((x + y) \leq 0)$, and constraints $(x \geq k1), (y \geq k2)$ are true, where k1 and k2 are constants. We have $((x + y) \leq 0) \wedge (x \geq k1) \wedge (y \geq k2) \Rightarrow (k1 + k2) \leq 0$. If we can show that $\neg((k1 + k2) \leq 0)$ holds, then we know that $\neg((x + y) < 0)$ holds.
We could not find any equivalent mutants that would be detected using the first case among our test programs. We found only one equivalent mutant that could be detected using the second case among the programs we used in our experiment. Thus, this technique was not implemented.

Better constraints

Since we use constraints to detect equivalent mutants, good constraints are needed to detect equivalent mutants. Right now, *Congen* only generates constraints on certain templates (listed in Appendix B), and it skips generation of complicated constraints. One limitation we found has to do with array constraints generation. Right now, *Congen* generates array constraints without indexes, which is safer when indexes are variables. But we could have array constraints with indexes when the indexes are constants. We analyzed the program *Cal* and found that if we could have array constraints with constant-indexes, *Equivalencer* could detect 69 more equivalent mutants, which would increase the detection percentage from 15.48% to 41.42%.
Another thought on improving the constraints is that of using humans to help with difficult constraints. We imagine an interactive system that interacts with the tester to get help with difficult constraints.

Analysis of the execution after the mutated statement

In this thesis, we only analyzed the execution up to and including the mutated statement. Following is a case where we could detect an equivalent mutant by analyzing the execution after the mutated statement. Figure 10 shows a segment of a program and a mutant this segment. If the domain of B is not limited to positive, and we only analyze the execution before the mutated statement, we are not able to detect that this mutant is equivalent. But by analyzing the execution after the mutated statement, we can detect the equivalence, since (A × A) is equivalent to (abs(A) × abs(A)). Analysis of the execution after the mutated statements involves analyzing sufficient conditions for an effective test case set. The sufficient condition was discussed in section 3.1. Currently, *Godzilla* provides no help with the sufficiency of test data.

6.2.2 Improving the software

Right now, the tool *Equivalencer* is separate from *Conform*. *Conform* originally was implemented to enhance the ability of generating test cases for *Godzilla*. Though *Conform* helps *Equivalencer* detect equivalent

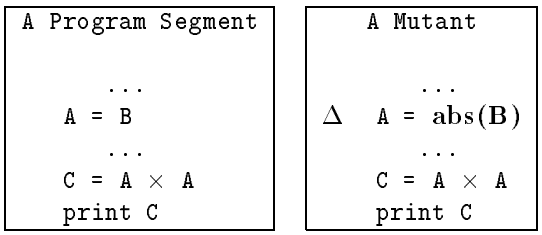

Figure 10: **Analysis After the Mutated Statement**

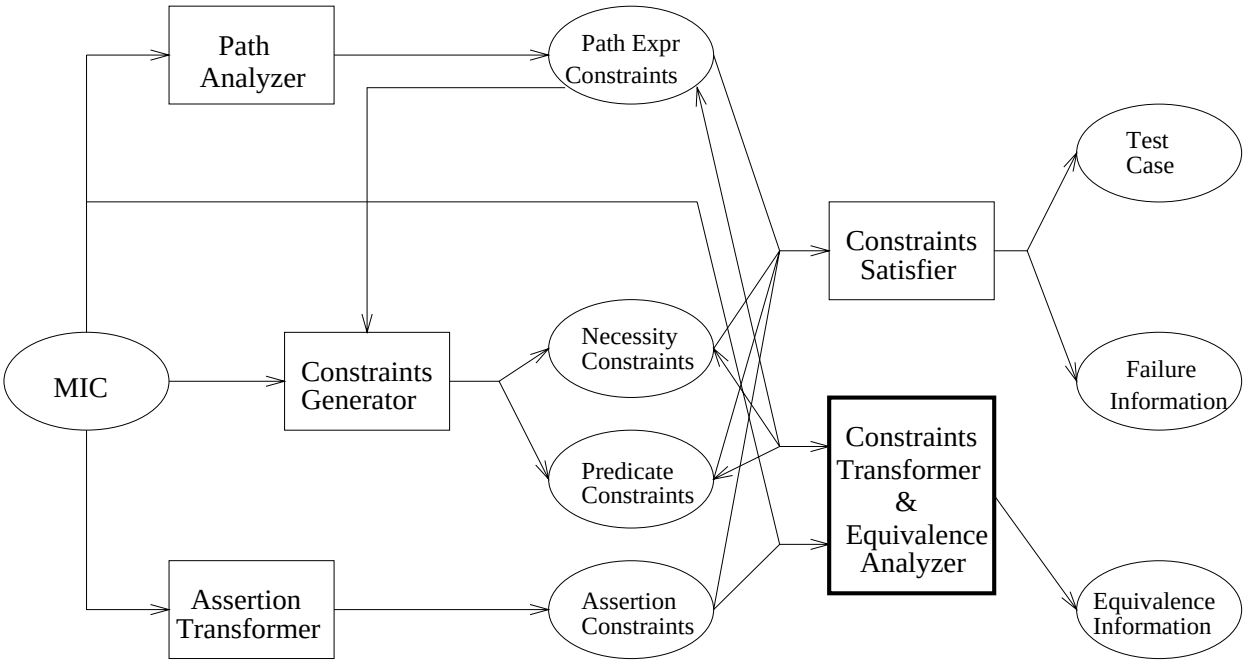

Figure 11: **Integration of Conform and Equivalencer with Godzilla**

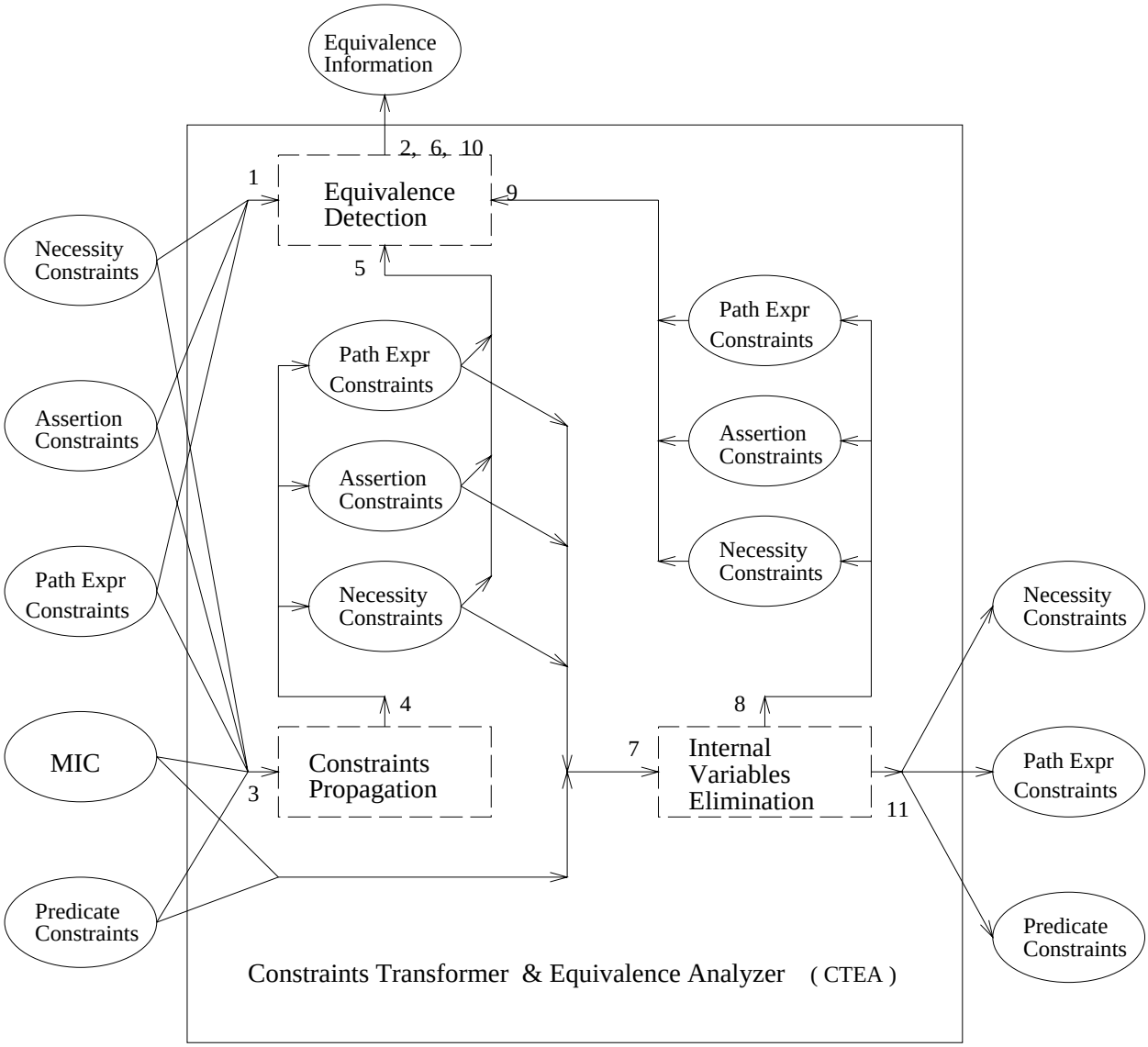

Figure 12: **Decomposition of Conform and Equivalencer's Integration**

mutants by propagating the constraints, it increases the difficulty of detection by throwing away considerable information that *Equivalencer* needs, such as internal variables. Also, *Conform* did not propagate the assertion constraints. That is why in the process described in section 5, we have to run *Transass* twice and *Equivalencer* twice.

If *Conform* and *Equivalencer* could be integrated together, we think it would increase the efficiency and effectiveness of detecting equivalent mutants. Figure 11 shows the architecture diagram of *Godzilla* after integrating *Conform* and *Equivalencer*, we refer to this as CTEA (Constraints Transformer & Equivalence Analyzer) later. Figure 12 shows a diagram of the decomposition of CTEA. The solid line boxes, ovals and arrows in the figures have the same meanings as in Figure 8. The dash line boxes represent the sub-functions under the CTEA. The numbers in the diagram show a process scenario of the CTEA.

First *Equivalence Detection* takes the original necessity, assertion and path expression constraints to detect equivalent mutants, (1, 2). Those constraints are shown on the left of the figure. *Constraints Propagation* propagates these constraints and re-writes them, (3, 4). They are shown in the left side within the big box of the figure. Then *Equivalence Detection* uses these propagated constraints to detect more equivalent mutants, (5, 6). Then, *Internal Variable Elimination* evaluates the constraints and re-writes them by using parameter variables instead of internal variables, (7, 8). They are shown in the right side within the big box of the figure. *Equivalence Detection* uses these constraints to detect more equivalent mutants again, (9, 10). Finally, *Internal Variable Elimination* outputs the necessity, path expression and predicate constraints, which are shown in the right of the figure, (11). Note that the necessity, path expression and predicate constraints outputted are the same as the ones in the right side within the big box. *Consat* will use these necessity and path expression constraints, along with the predicate constraints, to generate test cases.

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.

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

[How76] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.

[Hua75] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.

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

[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), 1994. Accepted for publication.

[Off88] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. Technical report GIT-ICS 88/28, (Also released as Purdue University Software Engineering Research Center technical report SERC-TR-25-P).

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

[OLR+94] A. J. Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. Technical report ISSE-TR-94-100, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1994. Under revision for ACM Transactions on Software Engineering Methodology.

[Som92] I. Sommerville. *Software Engineering.* Addison-Wesley Publishing Company Inc., 4th edition, 1992.

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

Appendix

Appendix A: Source Code of Programs

Function *Bsearch*

```
      LOGICAL Function BSEARCH (LIST, ELEM)
      INTEGER LIST (10), ELEM, LOW, HIGH, MID
C     Internal variables assertions:
      ASSERT (LOW.GT.0 .AND. HIGH.GE.0 .AND. LOW.LE.10 .AND. HIGH.LE.10)
      LOW = 1
      HIGH = 10
10    MID = (LOW + HIGH) /2
      IF (HIGH.LT.LOW) THEN
         BSEARCH = .FALSE.
         RETURN
      ELSE
         IF (ELEM.EQ.LIST(MID)) THEN
            BSEARCH = .TRUE.
            RETURN
         ELSE
            IF (ELEM.GT.LIST(MID)) THEN
               LOW = MID + 1
            ELSE
               HIGH = MID - 1
            ENDIF
            GOTO 10
         ENDIF
      ENDIF
      END
```

Function *Bub*

```
      SUBROUTINE BUBBLE (A)
      INTEGER A (5)
C=      A       inout
C     Internal variables assertions:
      ASSERT (I.GT.0 .AND. J.GT.0 .AND. N.GT.0 .AND.
     *        I.LE.5 .AND. J.LE.5 .AND. N.LE.5)
C     Sort A() using the bubble sort technique.

      INTEGER N, ITMP

      N = 5
      DO 200 J = N-1, 1, -1
         DO 100 I = 1, J, 1
            IF (A (I).LE.A (I+1)) GOTO 100
               ITMP = A (I)
               A (I) = A (I+1)
               A (I+1) = ITMP
100   CONTINUE
200   CONTINUE
      RETURN
      END
```

Function *Cal*

```
      INTEGER Function DAYS (DAY1, MONTH1, DAY2, MONTH2, YEAR)
      INTEGER DAY1, MONTH1, DAY2, MONTH2, YEAR
C     Calculate number of DAYS between the two given days.
C     DAY1 and DAY2 must be in same year.
C     Taken from Budd's thesis, pg 65, repeated from Geller [Gell78]
C     Translated from COBOL by Jeff Offutt, 3/88

C=    DAY1              in
C=    MONTH1    in
C=    DAY2              in
C=    MONTH2    in
C=    YEAR              in

      INTEGER DAYSIN (12), I
      INTEGER M4, M100, M400

C     Start range from 1-10000 for year.
C     External parameters assertions:
      ASSERT (DAY1.LE.31.AND.DAY2.LE.31.AND.
     *        MONTH1.LE.12.AND.MONTH2.LE.12 .AND.
     *        DAY1.GE.1 .AND. DAY2.GE.1 .AND.
     *        MONTH1.GE.1 .AND. MONTH2.GE.1 .AND. MONTH2.GE.MONTH1 .AND.
     *        YEAR.GE.1)
C     Internal variables assertions:
      ASSERT (I.GT.0 .AND. DAYSIN.GE.0 .AND. DAYS.GE.-2 .AND.
     *        M4.GE.0 .AND. M100.GE.0 .AND. M400.GE.0)
C     Internal variables assertions (hard to derive automatically):
      ASSERT (M4.LE.4 .AND. M100.LE.100 .AND. M400.LE.400)

C     If the dates are in the same month, we can
C     compute the number of days between them immediately.
      IF (MONTH2.EQ.MONTH1) THEN
         DAYS = DAY2 - DAY1
      ELSE
         DAYSIN (1) = 31
C        Are we in a leap year?
         M4   = MOD (YEAR, 4)
         M100 = MOD (YEAR, 100)
         M400 = MOD (YEAR, 400)
         IF ((M4.NE.0).OR.((M100.EQ.0).AND.(M400.NE.0))) THEN
            DAYSIN (2) = 28
         ELSE
            DAYSIN (2) = 29
         ENDIF
         DAYSIN (3) = 31
         DAYSIN (4) = 30
         DAYSIN (5) = 31
         DAYSIN (6) = 30
         DAYSIN (7) = 31
```

```
         DAYSIN (8) = 31
         DAYSIN (9) = 30
         DAYSIN (10) = 31
         DAYSIN (11) = 30
         DAYSIN (12) = 31

C        Start with days in the two months.
         DAYS = DAY2 + (DAYSIN (MONTH1) - DAY1)

C        Add the days in the intervening months
         DO 10 I = MONTH1+1, MONTH2-1, 1
            DAYS = DAYSIN (I) + DAYS
10       CONTINUE
      ENDIF
      RETURN
      END
```

Function *Euclid*

```
      INTEGER Function Euclid (A, B)
C
C     JEFF OFFUTT
C     12-07-89
C     Euclid's GCD algorithm.
C=      A       in
C=      B       in
C
C     Input vars.
      INTEGER A, B
C     Local vars.
      INTEGER Div, Rem
C     External parameters assertions:
      ASSERT (A .GT. 0 .AND. B .GT. 0)
C     Internal variables assertions:
      ASSERT (Div.GE.0 .AND. Rem.GE.0)

      Rem = 1
C     WHILE (Rem .GT. 0) DO
10    CONTINUE
      IF (Rem .LE. 0) GOTO 20
         Div = A/B
         Rem = A - Div*B
         A = B
         B = Rem
C     ENDWHILE
      GOTO 10
20    CONTINUE

      Euclid = A
      END
```

Function *Find*

```
      SUBROUTINE FIND (A, N, F)
      INTEGER A (10), N, F

C=  A  inout
C=  N  in
C=  F  in
C     F is index into A().  After execution, all elements to the left of
C     A(F) are less than or equal to A(F) and all elements to the right of
C     A(F) are greater than or equal to A(F).
C     Only the first N elements are considered.
C     From DeMillo, Lipton, and Sayward [DeMi78], repeated from Hoare's
C     paper [Hoar70].

      INTEGER M, NS, R, I, J, W

C     External parameters assertions:
      ASSERT (F.GE.1.AND.F.LE.N.AND.N.GE.1.AND.N.LE.10)
C     Internal variables assertions:
      ASSERT (I.GE.1.AND.M.GE.1)
C     Internal variables assertions (hard to derive automatically):
      ASSERT (J.GE.0 .AND. NS.GE.1)
      M = 1
      NS = N
10    IF (M.GE.NS) GOTO 1000
      R = A (F)
      I = M
      J = NS
20    IF (I.GT.J) GOTO 60
30    IF (A(I).GE.R) GOTO 40
      I = I + 1
      GOTO 30
40    IF (R.GE.A(J)) GOTO 50
      J = J - 1
      GOTO 40
50    IF (I.GT.J) GOTO 20
      W = A (I)
      A (I) = A (J)
      A (J) = W
      I = I + 1
      J = J - 1
      GOTO 20
60    IF (F.GT.J) GOTO 70
      NS = J
      GOTO 10
70    IF (I.GT.F) GOTO 1000
      M = I
      GOTO 10
1000  RETURN
      END
```

Function *Insert*

```
      SUBROUTINE INSERT (L,N)
C     Insertion sort on L.
      INTEGER  L(N),N
      INTEGER  KEY,I,J
C=  L inout
C=  N in

C     External parameters assertions:
      ASSERT (N.EQ.10)
C     Internal variables assertions:
      ASSERT (I.GE.0 .AND. J.GE.1)
C     By default, consat makes all adjustable arrays of length 10;
C     use the "-j"
C     option to change this default.

      J=2
1     IF (J.GT.N) GOTO 99
      KEY=L(J)
      I=J-1
5        IF (I.LE.0) GOTO 15
            IF (L(I).LE.KEY) GOTO 15
               L(I+1) = L(I)
               I=I-1
      GOTO 5
15    L(I+1) = KEY
      J=J+1
      GOTO 1
99    RETURN
      END
```

Function *Mid*

```
      Function MID (X, Y, Z)
      INTEGER X, Y, Z
C=      X       in
C=      Y       in
C=      Z       in

      MID = Z
      IF (Y.LT.Z) THEN
         IF (X.LT.Y) THEN
            MID = Y
         ELSE IF (X.LT.Z) THEN
            MID = X
         ENDIF
      ELSE
         IF (X.GT.Y) THEN
            MID = Y
         ELSE IF (X.GT.Z) THEN
            MID = X
         ENDIF
      ENDIF
      RETURN
      END
```

Function *Pat*

```
      INTEGER Function PAT (Subjct, Patern, SubLen, PatLen)
      INTEGER Subjct (SubLen), Patern (PatLen)
      INTEGER SubLen, PatLen

C=  Subjct  in
C=  Patern  in
C=  SubLen  in
C=  PatLen  in

C     This Function decides if the pattern (Patern) is found in
C     the subject (Subjct).
C     The position in the subject where the pattern starts is returned.
C     If the pattern is not in the subject, 0 is returned.
C     The pattern must be shorter than the subject.

      INTEGER SIndex, PIndex

C     External parameters assertions:
      ASSERT (PatLen.LE.SubLen .AND. PatLen.LE.10 .AND.
     *        SubLen.LE.10 .AND. PatLen.GE.1 .AND. SubLen.GE.1)
C     Internal variables assertions:
      ASSERT (SIndex.GE.1 .AND. PIndex.GE.1 .AND. Pat.GE.0)

C     Loop through the subject.  If the subject character equals the
C     first pattern character, check the rest of the pattern.

      Pat    = 0
      SIndex = 1

C     WHILE ( ) DO
20    CONTINUE
      IF (Pat .GT. 0 .OR. SIndex+PatLen-1 .GT. SubLen) GOTO 10
         IF (Subjct (SIndex) .EQ. Patern (1)) THEN
            Pat   = SIndex
            DO 30 PIndex = 1, PatLen
               IF (Subjct (SIndex+PIndex-1) .NE. Patern (PIndex)) THEN
                  Pat   = 0
               ENDIF
30          CONTINUE
         ENDIF
         SIndex = SIndex + 1
      GOTO 20
C     ENDWHILE
10    CONTINUE

      RETURN
      END
```

Function *Quad*

```
C
C     JEFF OFFUTT
C     12-02-89
C
C     THIS PROGRAM ACCEPTS IN THREE CONSTANTS A, B, AND C THAT
C     REPRESENT QUADRATIC EQUATIONS.  FOR EACH EQUATION, THE
C     ROOTS ARE COMPUTED AND PRINTED IF THEY EXIST.
C
      SUBROUTINE ROOTS (CoeffA, CoeffB, CoeffC, Root1, Root2)
C     INPUT VARIABLES ...
      REAL CoeffA, CoeffB, CoeffC
C=  CoeffA  in
C=  CoeffB  in
C=  CoeffC  in

C     OUTPUT VARIABLES ...
      REAL Root1, Root2
C=  Root1  out
C=  Root2  out

C     Internal VARIABLES ...
      REAL Disc
      ASSERT (CoeffA .NE. 0.0)

      Disc = CoeffB*CoeffB - (4*CoeffA*CoeffC)
      IF (DISC .GE. 0.0) THEN
         Root1 = ((-CoeffB) + SQRT(Disc)) / (2*CoeffA)
         Root2 = ((-CoeffB) - SQRT(Disc)) / (2*CoeffA)
      ELSE
         Root1 = 0.0
         Root2 = 0.0
      ENDIF
      STOP
      END
```

Function *Trityp*

```
       INTEGER Function TRIANG(I,J,K)

       INTEGER I,J,K

C=  I in
C=  J in
C=  K in

C      MATCH IS OUTPUT FROM THE ROUTINE:
C         TRIANG = 1 IF TRIANGLE IS SCALENE
C         TRIANG = 2 IF TRIANGLE IS ISOSCELES
C         TRIANG = 3 IF TRIANGLE IS EQUILATERAL
C         TRIANG = 4 IF NOT A TRIANGLE

C      After a quick confirmation that it's a legal
C      triangle, detect any sides of equal length

C      Internal variables assertions:
       ASSERT (TRIANG.GE.0 .AND. TRIANG.LE.6)
       IF (I.LE.0.OR.J.LE.0.OR.K.LE.0) THEN
          TRIANG=4
          RETURN
       ENDIF
       TRIANG=0
       IF (I.EQ.J) TRIANG=TRIANG+1
       IF (I.EQ.K) TRIANG=TRIANG+2
       IF (J.EQ.K) TRIANG=TRIANG+3
       IF (TRIANG.EQ.0) THEN

C      Confirm it's a legal triangle before declaring
C      it to be scalene

          IF (I+J.LE.K.OR.J+K.LE.I.OR.I+K.LE.J) THEN
             TRIANG = 4
          ELSE
             TRIANG = 1
          ENDIF
          RETURN
       ENDIF

C      Confirm it's a legal triangle before declaring
C      it to be isosceles or equilateral

       IF (TRIANG.GT.3) THEN
          TRIANG = 3
       ELSE IF (TRIANG.EQ.1.AND.I+J.GT.K) THEN
          TRIANG = 2
       ELSE IF (TRIANG.EQ.2.AND.I+K.GT.J) THEN
          TRIANG = 2
```

```
ELSE IF (TRIANG.EQ.3.AND.J+K.GT.I) THEN
   TRIANG = 2
ELSE
   TRIANG = 4
ENDIF

END
```

Function *Warshall*

```
      SUBROUTINE WARSHALL (A)
      INTEGER A (5,5)
C=        A     inout

C     Calculate the transitive closure of A using Warshall's algorithm.

C     External parameters assertions:
      ASSERT (A.LE.1.AND.A.GE.0)
C     Internal variables assertions:
      ASSERT (I.GE.1 .AND. J.GE.1 .AND. K.GE.1 .AND.
     *        I.LE.5 .AND. J.LE.5 .AND. K.LE.5)
       INTEGER I, J, K

       DO 100 K = 1, 5
          DO 200 I = 1, 5
             DO 300 J = 1, 5
                IF (A(I, J) .EQ. 0) THEN
                   A(I, J) = A(I, K) * A(K, J)
                ENDIF
300          CONTINUE
200       CONTINUE
100    CONTINUE
       RETURN
       END
```

Appendix B: Template of Constraints

Type	Description	Constraint
aar	array for array replacement	$A(e_1) \neq B(e_2)$
abs	absolute value insertion	$e_1 < 0$
		$e_1 > 0$
		$e_1 = 0$
acr	array constant replacement	$C \neq A(e_1)$
aor	arithmetic operator replacement	$e_1\ \rho\ e_2 \neq e_1\ \phi\ e_2$
		$e_1\ \rho\ e_2 \neq e_1$
		$e_1\ \rho\ e_2 \neq e_2$
		$e_1\ \rho\ e_2 \neq Mod(e_1, e_2)$
asr	array for variable replacement	$X \neq A(e_1)$
car	constant for array replacement	$A(e_1) \neq C$
cnr	comparable array replacement	$A(e_1) \neq B(e_2)$
csr	constant for scalar replacement	$X \neq C$
der	DO statement end replacement	$e_2 \Leftrightarrow e_1 \geq 2$
		$e_2 \leq e_1$
lcr	logical connector replacement	$e_1\ \rho\ e_2 \neq e_1\ \phi\ e_2$
ror	relational operator replacement	$e_1\ \rho\ e_2 \neq e_1\ \phi\ e_2$
sar	scalar for array replacement	$A(e_1) \neq X$
scr	scalar for constant replacement	$C \neq X$
svr	scalar variable replacement	$X \neq Y$

Table 13: **Constraint Templates**

Appendix C: Mutation Operators Used in Mothra

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 14: **Mothra Mutation Operators**