Automatic Test Data Generation

Andrew Jefferson Offutt VI

A Thesis
Presented to
The Faculty of the Division of Graduate Studies

in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in the School of Information and Computer Science
Georgia Institute of Technology

August 1988

(Released as technical report GIT-ICS 88/28)

Copyright © 1988 by A. J. Offutt. All rights reserved.
ACKNOWLEDGEMENTS

After any long hard effort, one of the most pleasurable duties is always to thank the people who helped make it possible. My list is long and must begin with my advisor, Professor Richard DeMillo. Thanks for the long years of support, leading me by the nose when I was confused, and describing the problem that this research was trying to solve. He certainly taught me more than he has realized.

I also would like thank the reading committee, Drs. Bill Appelbe, Jim Burns, and Bob Horgan for their suggestions, their interest and their time spent reading the dissertation. Special thanks goes to Bill Appelbe and Jim Burns for functioning as "advisors in absentia" during the last year of this dissertation. Without their advice and support, both technical and emotional, this work may never have been completed. I also appreciate Jim Burns help with Chapter IV.

The members of the MOTHRA project have been a marvelous source of positive criticism, advice, support, and ideas. Thanks goes to Mike McCracken for never letting me fool myself and to Ronnie Martin for her close criticism. I especially appreciate Ed Krauser for sharing the production of MOTHRA, invaluable help with the dissertation, and for showing me the meaning of hard work. Dr. Kim King encouraged me early and often in my graduate work and taught me some measure of precision. Thanks also goes to Dr. Gene Spafford and Daney Guidi for long friendships. The MOTHRA group has been invaluable to my education and my only regret is that I fear I will never have the opportunity to work with as dynamic and productive a group again.

Other faculty and students of the School of Information and Computer Science have been particularly helpful. I particularly appreciate the great amounts of advice, help and friendship from Sinai Bilgel, who took me under his wing in my early days in graduate school. Thanks go to Dr. Dave Pitts, Dr. Dana Eckhart, Dr. Tom Wilkes and Morgan Morris for examples. My appreciation goes to Dr. Rich LeBlanc, John Passafiume, Dr. Jerry Spinrad and Dr. Ian Akyildiz. Scott Vorthman has suggested several good ideas to me, including the concept of using constraints to integrate testing with software development environments. For his friendship and assistance, for the unflagging support of my many moods, and for helping me with paper work while I was away from Georgia Tech, thanks go to my office mate, Walt Ligon.

After spending two extended visits at Purdue University, I certainly appreciate the hospitality, help, and courtesy extended to me by the members of the Computer Science department and the Software Engineering Research Center. In addition to the members of the MOTHRA project mentioned above, I thank Hira Agrawal, Gui Cabral, Kathy Heaphy, William Hsu, Wynne Hsu, Aditya Mathur, and Elizabeth Northern for helping to make my visits both productive and enjoyable.

I would also like to give my appreciation to the INFOMANIACS for comedy relief, to the Atlanta Hash House Harriers and Harriettes for relief, and to the many denizens of SAC; with whom I've escaped many times for games of hard-fought hoops. I also appreciate Amy Pitts for demonstrating the art of winning at bridge.

The many people who have helped me get to this point were invaluable and are innumerable. Most importantly were Mary Alice Jayne for not only teaching me, but encouraging me to read, Dr. Glenn Johnston for telling me that I was a "real mathematician", and to Dr. Richard K. Lyon for giving me a chance.

Most of all I wish to thank my wife Kathy for hugs, for organizing our lives, and most importantly for waiting.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>viii</td>
</tr>
<tr>
<td>I. SOFTWARE TESTING</td>
<td>1</td>
</tr>
<tr>
<td>Software Testing Techniques</td>
<td>2</td>
</tr>
<tr>
<td>Reliability of Test Data</td>
<td>13</td>
</tr>
<tr>
<td>II. CONSTRAINT BASED TESTING CONCEPTS</td>
<td>15</td>
</tr>
<tr>
<td>Killing Mutants</td>
<td>15</td>
</tr>
<tr>
<td>Paths and Constraints</td>
<td>16</td>
</tr>
<tr>
<td>Constraint Based Testing</td>
<td>19</td>
</tr>
<tr>
<td>III. NECESSITY CONSTRAINTS</td>
<td>24</td>
</tr>
<tr>
<td>Mutant Operator Constraints</td>
<td>24</td>
</tr>
<tr>
<td>Predicate Constraints</td>
<td>28</td>
</tr>
<tr>
<td>IV. COMPUTING PATH EXPRESSIONS</td>
<td>29</td>
</tr>
<tr>
<td>V. CONSTRAINT SATISFACTION</td>
<td>34</td>
</tr>
<tr>
<td>Definitions</td>
<td>34</td>
</tr>
<tr>
<td>Constraint Satisfaction Methods</td>
<td>35</td>
</tr>
<tr>
<td>Satisfaction of Test Case Constraints</td>
<td>40</td>
</tr>
<tr>
<td>VI. INTERNAL VARIABLES</td>
<td>50</td>
</tr>
<tr>
<td>Solving for Internal Variables</td>
<td>50</td>
</tr>
<tr>
<td>Internal Variable Experimentation</td>
<td>54</td>
</tr>
<tr>
<td>VII. DETECTING EQUIVALENT MUTANTS</td>
<td>56</td>
</tr>
<tr>
<td>Budd’s Equivalent Mutant Difficulty Levels</td>
<td>56</td>
</tr>
<tr>
<td>Detecting Equivalent Mutants By Hand</td>
<td>58</td>
</tr>
<tr>
<td>Using Compiler Optimization Techniques to Detect Equivalent Mutants</td>
<td>58</td>
</tr>
<tr>
<td>Using Constraints to Detect Equivalent Mutants</td>
<td>61</td>
</tr>
<tr>
<td>VIII. IMPLEMENTATION OF A CONSTRAINT BASED GENERATOR</td>
<td>63</td>
</tr>
<tr>
<td>Architecture of a Test Case Generation System</td>
<td>63</td>
</tr>
<tr>
<td>Auxiliary Test Data Generation Tools</td>
<td>63</td>
</tr>
<tr>
<td>IX. EMPIRICAL STUDIES</td>
<td>69</td>
</tr>
<tr>
<td>The Program Suite</td>
<td>70</td>
</tr>
<tr>
<td>Effectiveness of Technique</td>
<td>70</td>
</tr>
<tr>
<td>Performance of Technique</td>
<td>75</td>
</tr>
<tr>
<td>Improvement of Technique Effectiveness</td>
<td>77</td>
</tr>
<tr>
<td>Improvement of Technique Performance</td>
<td>77</td>
</tr>
</tbody>
</table>
X. CONCLUSIONS AND SUGGESTIONS FOR FURTHER STUDIES ........................................... 82
   Most Effective Modes of Operation ................................................................. 82
   Satisfaction of Constraints ............................................................................. 83
   Testing Program Specifications ..................................................................... 83
   Join with Software Development Environments .......................................... 84
   Extending Equivalent Muam Detection ........................................................ 85
   Summary ........................................................................................................... 85

BIBLIOGRAPHY ...................................................................................................... 87

APPENDIX A ........................................................................................................ 92

APPENDIX B ........................................................................................................ 105

APPENDIX C ........................................................................................................ 112

VITA ...................................................................................................................... 141
LIST OF ILLUSTRATIONS

Figure 1. Function MAX ........................................... 5
Figure 2. Error-Based Testing ...................................... 12
Figure 3. Mutant of BSEARCH ...................................... 16
Figure 4. Sufficiency Example ...................................... 19
Figure 5. Input Domain Subsets .................................... 21
Figure 6. Expanded Input Domain Subsets ...................... 22
Figure 7. Predicate Problem ........................................ 28
Figure 8. Path Expression Algorithm ............................ 31
Figure 9. Back-Branching Problem ............................... 32
Figure 10. BSEARCH Path Expressions ......................... 33
Figure 11. Propagation Procedure ............................... 37
Figure 12. Dependency Graph ..................................... 39
Figure 13. Random Satisfaction Procedure .................... 41
Figure 14. Propagate and Guess procedure ................... 43
Figure 15. Bisection Procedure ................................... 44
Figure 16. Domain Reduction Procedure ....................... 47
Figure 17. Internal Variable Expression Algorithm .......... 52
Figure 18. Internal Variable Resolution Algorithm .......... 53
Figure 19. Common Subexpression Example ................... 59
Figure 20. Loop Invariants Optimization Example .......... 60
Figure 21. Hoisting Optimization Example .................... 60
Figure 22. Hoisted Version ........................................ 60
Figure 23. Equivalence Detection ............................... 61
Figure 24. Godzilla Automatic Generator Architecture ...... 64
Figure 25. Godzilla Architecture—Auxiliary Tools .......... 65
Figure 26. Assertion in Function MAX .......................... 66
Figure 27. BSEARCH Equivalent Table Example ............. 67
Figure 28. Array Version of Max—Specification Example .... 84
Figure 29. Results of Specification Example .................. 84
LIST OF TABLES

Table 1. Mothra Mutant Operators ........................................... 4
Table 2. Growth of F ................................................................. 23
Table 3. Replacement Mutants .................................................. 25
Table 4. Constraint Templates—Replacement Mutants ..................... 25
Table 5. Constraint Templates—Constraint Replacement Mutants ........ 26
Table 6. Constraint Templates—Operator Mutants .......................... 26
Table 7. Constraint Templates—Insertion Mutants .......................... 27
Table 8. Constraint Templates—Statement Mutants ....................... 27
Table 9. Path Expression Rules .................................................. 30
Table 10. Random Satisfaction Results for TRITYP ............................. 42
Table 11. Internal Variable Experiment ....................................... 54
Table 12. Equivalent Mutant Percentages ..................................... 57
Table 13. Percentages of Equivalent Mutants by Level ..................... 58
Table 14. Test Case Adequacy Results ....................................... 71
Table 15. Effectiveness Experiment Results .................................. 72
Table 16. Summary of Comparison Results ................................... 73
Table 17. Equivalent Mutants Coefficients ................................. 75
Table 18. Test Data Generation Times ........................................ 76
Table 19. Speedup of Automated Method ..................................... 76
Table 20. Internal Variable Comparison ..................................... 77
Table 21. Test Cases Efficiency ................................................. 79
Table 22. Combining Constraints Results .................................... 79
Table 23. Reduction Coefficient of Combining Constraints Experiments 80
Table 24. Number of Satisfaction Attempts ................................... 81
Table 25. Satisfaction Attempts Versus Constraints .......................... 81
SUMMARY

This dissertation presents a completely automatable test data generation technique based on the notion of adequacy. The technique has been implemented as part of the Mothra mutation system.

Software testing research has often focused on developing techniques to generate test data. Most techniques attempt to generate test data that is reliable, where for a reliable test data set, if the program is correct on the test data, then it is correct on all possible inputs. There are several disadvantages with generating test data based on reliability. First, generating reliable test data sets is an undecidable problem and there is no theoretical way of weakening reliability. Second, for each of these reliability based techniques, there are examples of relatively simple programming errors that will not be found by the test data generated for that technique. Third, most of these techniques have not been successively automated. Specifically, there are known ways for generating descriptions of test cases based on the criteria for each of the techniques, but there has been no method for translating these descriptions into test data.

Test data adequacy is another criteria for generating test data. A test data set is adequate for a program if the set distinguishes a correct version of the program from every incorrect version of the program. Although generating adequate test sets is also undecidable, there is a known method for weakening adequacy whose theoretical properties are well understood. Relative-adequacy, also known as mutation-adequacy, requires that the test data set distinguish a correct version of the program from a finite number of incorrect versions of the program. This dissertation presents a completely automatable technique for generating test data sets that approximate mutation-adequacy. The technique defines conditions under which mutant programs will die that are represented as mathematical constraints that are automatically satisfied to generate sets of data.

The major results of this dissertation include a new technique for automatically generating test data that can be used to effectively kill mutants in a mutation system (specifically the Mothra testing system), and that also combines the capabilities of previous test data generation methods. Also, procedures for creating mathematical constraints that describe the test cases and for solving those constraints to produce test cases are presented. This technique has been implemented as a prototype test data generation tool (Godzilla) that is integrated with the Mothra mutation testing system.

Experimentation with the Godzilla generator and the Mothra system has indicated that this technique produces test data that is closer to adequate than other testing techniques. Based on the evidence presented in the dissertation, the technique is at least competitive with other test data techniques and may well be more powerful. Also, the technique is easy to extend to detect more errors or incorporate the capabilities of new testing techniques. Perhaps most importantly, the technique is easily automatable, as demonstrated by the prototype generator Godzilla.
CHAPTER I
SOFTWARE TESTING

If you build a better mouse trap
you will catch better mice.
— George Gobel

In 1964, NASA's Gemini V spacecraft splashed down nearly 100 miles from its planned destination. Subsequent investigation revealed a problem with the navigation software. A programmer had tried to simplify a calculation by ignoring the fact that the Earth rotates completely every day. He assumed that any particular spot on the Earth's surface would rotate to the same location in space every twenty-four hours. Unfortunately, the Earth also rotates about the sun, which means that the landing site was not in the same place every day. Luckily, the capsule still splashed down in the ocean, although it was "lost" for several hours [Fox83].

The problem with the Gemini V mission was a dramatic failure that certainly could have been worse if the capsule had come down on land or if the mission had lasted longer (putting the spaceship further off course). This kind of problem can (and does) exist in any piece of software and is only exaggerated with software that controls machines in situations where human lives are at stake. Whenever we use software we risk some failure. The consequences of a failure can range from humorous to disastrous, but we would certainly like to find ways to reduce the amount of risk of using software.
That is the main purpose of software testing—to serve as a risk reducing activity. Of course we cannot totally eliminate risk by determining that a program is correct, but we can reduce the risk of using software by finding and eliminating faults. Reducing the risk of using a piece of software also increases our confidence that the software will perform as intended. Increasing our confidence in the software is another important goal of software testing. Software testing is performed by choosing inputs, called test cases, and executing the program on the test cases to determine if the program is correct on that particular input. By doing this, software testing attempts to provide a partial answer to the following question:

If a program is correct on a finite number of test cases, is it correct in general?

This dissertation describes a new technique for automatically generating test data that is assured to be effective at detecting errors. The test data is created from constraints that describe effective test cases. The technique is called "Constraint Based Test Data Generation" (CBT). The remainder of this chapter introduces software testing and test data generation, and outlines the current research in the area both from a practical and a formal point of view. Several classes of testing techniques are surveyed. One of these techniques is mutation analysis. Throughout this dissertation, the rationale for CBT is presented in terms of mutation analysis, as the two methods share the same underlying theoretical basis. The implementation of this dissertation is part of the Mothra Software Testing Environment that is being developed jointly at Georgia Tech and Purdue University [DeMan, DeM86, DeM89, Bud87]. This implementation is called Godzilla and has been integrated with version 1.2 of the Mothra testing system [DeM87]. The Mothra system is the latest in a long series of testing systems that have implemented mutation on various machines and for various languages [Budd78, Budd77, Hank80].

Chapters II and III develop a model for generating test data based on the concept of test case adequacy. Chapters IV, V, VI and VII explore the problems and present possible solutions involved in generating adequate test data. Chapter VIII describes an implementation of constraint based testing (the Godzilla test data generator) and Chapter IX presents a series of experiments designed to demonstrate the usefulness of the technique and to explore methodologies for applying the technique. Chapter X reviews the major results of the dissertation and presents some avenues for future development and uses of CBT.

Software Testing Techniques

A major goal of software testing research has been to develop techniques for generating test data sets that are finite yet also gives the tester some confidence in the software being tested. Of course, a correct program should inspire complete confidence in that program. In software testing, we often judge program correctness by use of an oracle. An oracle can determine whether any particular output of a program is correct on that input. An oracle can be the specifications for the program, an automated system, or, as is usually the case in practice, the tester. A correct program computes correct output (as judged by the oracle) on every input case. Since determining program correctness through testing is, as a rule, impractical, testing techniques attempt to generate test data that meet certain criteria. This section surveys some of the more common of these techniques. A more complete survey can be found in [DeM87a]; the techniques described here are chosen because as a group they provide a perspective for the technique presented in this dissertation.

Random Testing

Test data generation techniques choose test data from the input domain of the program according to some selection criteria. In random testing, the data points are chosen randomly [Dura81]. Various

---

1. This important and basic result is due to Howden [How76] and is reviewed later in this chapter.
2. "Correctness" will be defined formally later in this chapter.
probability distributions can be chosen to decide which test cases are chosen. Some advantages of random testing are that the technique is simple to describe and easy to automate. The ability to use different probability distributions also allows a tester to choose the strength of the test data produced by random testing. Although random testing has been shown to produce test cases that are not as effective at finding faults as techniques that use structural information of the program [DeMi78, Hetz76, Girg86, DeMi78a], as Duran and Niaios [Duran81] point out, the relative inexpensiveness of this approach often makes it cost effective.

Functional Testing

Since software is written to implement specific functions, it seems logical to construct test data that specifically tests the functions that the software implements. There are two steps in functional testing. First the program is decomposed into functional units that correspond to modules identified in the design of the program. Then test data is generated that tests the functional units independently [Howd87]. The data can be derived from formal specifications, formal design, or informal descriptions of the system. It is very difficult to use functional testing techniques to automatically create test data [DeMi87a]. Also, like any black box method, functional testing is not effective at finding faults that arise from complicated code eccentricities [Meyer79].

Path Testing

A path through a program is a sequence of statements that define a possible flow of control through the source code. These paths partition the input domain of the program into domain subsets where each subset contains all the test cases that will cause the program to follow a particular control flow. If a program contains loops, then each iteration through a loop gives a different path. This means that we can, in theory, have an infinite number of paths through a program and an infinite number of domain subsets from which to choose test data.

Because of the large number of paths in a program, the obvious test data selection criteria of executing all paths (full path analysis) is infeasible. There are several ways to relax this criteria, as described by Clarke et al [Clar85]. The simplest form is statement analysis, where each statement is required to be executed once. Although statement analysis is simple to understand and easy to automate, simple examples can be constructed where failures occur only if a particular sequence of statements are executed [DeMi78]. Statement analysis, of course, is not likely to find these faults.

A stronger requirement is to require that the test data execute each branch in the program at least once, where a branch is taken to be any deviation from sequential control flow through the program. This is called branch analysis. Again, although this is certainly necessary to thoroughly test a program, fairly simple faults can be found that branch analysis will not find. A still stronger version of path analysis is to ignore loops and to require that every non-iterative path be executed at least once.

In [Clar85] is a survey of various path analysis testing criteria. A hierarchy of techniques (with full path analysis as the most comprehensive and statement analysis the weakest) is presented in some detail.

Error Based Testing

DeMillio, Lipton and Sayward [DeMill81] introduced the concept of test data adequacy, and it was formally defined by Budl and Angluin [Budl82]. A test set is adequate for a program if correct versions of the program produce correct output for every case in the test set and if incorrect versions of the program produce incorrect output on at least one test case (adequacy is defined formally later in this chapter). Testing techniques that are based on adequacy (such as mutation analysis) produce test data that expose faults that are likely to occur in the program. That is, rather than trying to test for correctness, an adequacy based testing technique attempts to demonstrate that certain faults are not present in the program.
The term error based testing is due to Ostrand and Weyuker [OstrQ]. They presented a technique for generating test data that partitions a program's input domain into a set of disjoint classes called subdomains. These subdomains are chosen to reveal certain errors. If a subdomain is revealing for an error, then all members of the subdomain will reveal the error. So if the program is correct for one input from a subdomain that is revealing for that error, then that error can be eliminated. Unfortunately, Weyuker and Ostrand's approach does not specify the errors that can be eliminated.

Mutation analysis is an error based testing technique due to DeMillo, Lipton and Sayward [DeMi78]. Morell [More84] uses mutation analysis as a basic model for a theoric treatment of error based testing and shows that mutation analysis can be extended to include any error based testing technique.

Strictly speaking, mutation analysis is a test data measuring technique rather than a testing technique. Mutation analysis measures the quality of a set of externally created test cases [DeMi86, DeMi78, DeMi79], and the testing of the software is technically a side-effect of this measurement. In practice, a tester interacts with an automated mutation system to determine the adequacy of the current set of test data and to improve that test data. This forces the tester to test for specific types of faults. These faults are represented by a set of simple syntactic changes to the test program. Hence the terminology; these changed programs are mutants of the original and a mutant is killed by distinguishing the output of the mutant from that of the original program.

The complete set of mutant operators used by the Mithra mutation system is shown in Table 1. Each of the 22 mutant operators is referred to by a 3-digit acronym. For example, the uar mutant operator stands for an "array for array replacement" and causes each reference to an array to be replaced by each other distinct array reference. These operators are more completely described in the Mithra Internal Documentation [DeMi87] and will be analyzed individually in Chapter III.

<table>
<thead>
<tr>
<th>uar</th>
<th>array reference for array reference replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>absolute value insertion</td>
</tr>
<tr>
<td>acr</td>
<td>array reference for constant replacement</td>
</tr>
<tr>
<td>aor</td>
<td>arithmetic operator replacement</td>
</tr>
<tr>
<td>asr</td>
<td>array reference for scalar variable replacement</td>
</tr>
<tr>
<td>car</td>
<td>constant for array reference replacement</td>
</tr>
<tr>
<td>cnr</td>
<td>comparable array name replacement</td>
</tr>
<tr>
<td>crp</td>
<td>constant replacement</td>
</tr>
<tr>
<td>csr</td>
<td>constant for scalar replacement</td>
</tr>
<tr>
<td>der</td>
<td>DO statement end replacement</td>
</tr>
<tr>
<td>dsa</td>
<td>data statement alterations</td>
</tr>
<tr>
<td>glr</td>
<td>goto label replacement</td>
</tr>
<tr>
<td>lcr</td>
<td>unsigned short connector replacement</td>
</tr>
<tr>
<td>ror</td>
<td>relational operator replacement</td>
</tr>
<tr>
<td>rsr</td>
<td>return statement replacement</td>
</tr>
<tr>
<td>san</td>
<td>statement analysis (replacement by TRAP)</td>
</tr>
<tr>
<td>sar</td>
<td>scalar variable for array reference replacement</td>
</tr>
<tr>
<td>scrr</td>
<td>scalar for constant replacement</td>
</tr>
<tr>
<td>srl</td>
<td>statement deletion</td>
</tr>
<tr>
<td>src</td>
<td>source constant replacement</td>
</tr>
<tr>
<td>svr</td>
<td>scalar variable replacement</td>
</tr>
<tr>
<td>uoi</td>
<td>unary operator insertion</td>
</tr>
</tbody>
</table>

Table 1. Mithra Mutant Operators
When a program is submitted to a mutation system, the system first creates a set of mutated versions of the program. Next, the tester supplies a set of test data to the system that serves as inputs to the program. Each of these test cases is executed on the original program and the tester verifies that the output is correct. If the original program is correct on that test data, the test data is executed on each of the mutant programs in turn. If the output of the mutant program differs from the original output, then that mutant is marked dead. Dead mutants are not executed against subsequent test cases.

For example, function MAX\(^3\) is shown with four embedded mutants (lines preceded by a "#") in Figure 1. The convention when displaying mutants of a program is to show the original program with the mutated statements shown immediately below the original statements. Each of these mutated statements represents a different (mutant) program where the mutated statement replaces the statement immediately preceding it. MAX returns the larger of its two integer inputs:

```
FUNCTION MAX (M,N)
1   MAX = M
2   IF (N.GT.M) MAX = N
3   RETURN
```

Figure 1. Function MAX

Once all test cases have been executed, a mutation score is computed. The mutation score is the ratio of non-equivalent mutants. An equivalent mutant is a mutant that produces a functionally equivalent version of the original program. The last mutant in Figure 1 is an equivalent mutant because it does not matter whether the comparison is a strict greater than comparison. So if two of the mutants in Figure 1 were killed, the number of non-equivalent mutants is three, and the mutation score would be two divided by three; or 0.66. Equivalent mutants will be discussed in more detail later in this chapter and a technique for automatically detecting some equivalent mutants is presented in Chapter VII. The tester's goal is to raise the mutation score to 1.00, indicating that all mutants have been detected.

If (as is likely) mutants are alive after all mutants have been executed against the current test data set, the tester can enhance the set of test data by supplying new inputs. Note that even if the tester has not found faults with the current set of test cases, the mutation score indicates how well the program has been tested. Moreover, the live mutants point out the inadequacies in the test cases. In most cases, the tester creates test cases to kill specific mutants that are still alive. The process of adding new test cases, verifying output correctness, and killing mutants is repeated until the tester is satisfied with the mutation score. A mutation score threshold can be set as a policy decision to require testers to test software to a predefined level [DeMi80].

When examining mutation analysis it is easy to be confused about exactly when faults are detected. Faults in the software are detected when test cases are executed against the original program [DeMi78]. The tester must verify that the output of the program on each test case is correct. If the output is correct, the process continues as described. If the output is incorrect, then an error is found and the process stops until the error can be corrected. In practice, if the software contains a fault, there

---

3: Since the implementation of the constraint based technique (as part of the Mythra mutation system) test programs written in Fortran 77, this and all other examples in this dissertation are presented in Fortran 77.
will usually be a set of mutants that can only be killed by a test case that also detects that fault. The reasoning behind this statement is both theoretical and experimental. This is how a mutation system guides the tester into finding faults in the program. The tester will not be able to kill all mutants without finding most of the faults.

Before explaining the principles behind mutation analysis, the ideas can be made more concrete through an example run of the Mothra system. This example was generated by capturing the output to the screen from the CDemo interface to Mothra version 1.2. Some editing was necessary to remove control characters and to insert descriptive text. Also, some of the interaction was removed to avoid clutter and user inputs were bold-faced. Rather than trying to explain every nuance of the interaction being demonstrated, only the pertinent highlights are explained here. Refer to the Mothra User Manual [Bud87] for more details.

The tester first initiates the mutation experiment by executing the program cdemo, supplying an identifying name for the experiment (AJO) and supplying the path to the test program. cdemo then parses the program and presents the main Mothra menu.

```
> cdemo

MOTHRA.

Fortran-77 Mutation System

CDemo Interface
Version 1.2

Please enter the experiment name: AJO
Initial experiment...
Please enter the Fortran-77 source file name(s): -ofut/exp/src/max.f

Parsing source files...
Ready to start a mutation experiment ...

==========
CDemo Main Menu
==========

Enter the number of your choice.
1) Enable Mutants
2) Enter Test Case
3) Execute Mutants
4) Status Menu
5) View Live Mutants
6) Test Case Menu
7) View Program Source
8) Execute Mutants in Background
9) Exit CDemo
? 1
```
The tester first enables a set of mutants. For this example, the types rov and srv were used.

---

Mutant Names

**TYPES:**

aar abs acr acr asr asr ccr ccr cpr crp crp der dsl

glr lcr rov ror svr san sar src sll src srv uol

**CLASSES:**

ary con cpl dinm opm prd sel sm

**SUPER CLASSES:** all cca pda sal

Which mutant types/classes would you like to generate?
Enter any number of mutants on one line separated by blanks.
Enter ? to see all valid selections, or q to quit.

> ? rov srv

Creating mutants ...
Created 14 additional mutants for AIO

---

The tester then might wish to view the mutants that were created:

```
FUNCTION MAX (M, N)
MAX = M
# srv 7 # N = M
# srv 8 # MAX = N
  IF (N, GT, M) MAX = N
# srv 9 # IF (M, GT, M) MAX = N
# srv 10 # IF (MAX, GT, M) MAX = N
# srv 11 # IF (N, GT, N) MAX = N
# srv 12 # IF (N, GT, MAX) MAX = N
# ror 1 # IF (N, L,T, M) MAX = N
# ror 2 # IF (N, LE, M) MAX = N
# ror 3 # IF (N, EQ, M) MAX = N
# ror 4 # IF (N, NE, M) MAX = N
# ror 5 # IF (N, GE, M) MAX = N
# ror 6 # IF (.TRUE.,) MAX = N
# srv 13 # IF (N, GT, M) M = N
# srv 14 # IF (N, GT, M) MAX = M
RETURN
END
```

The tester examines these mutants and determines that the first mutant will be killed if the larger of the two numbers is \( N \). So he inputs the test case \( M = 1, N = 2 \):
CDemo Main Menu

Enter the number of your choice.
1) Enable Mutants
2) Enter Test Case
3) Execute Mutants
4) Status Menu
5) View Live Mutants
6) Test Case Menu
7) View Program Source
8) Execute Mutants in Background
9) Exit CDemo
?

Enter INTEGER value for M: 1
Enter INTEGER value for N: 2

Test case # 1

Running original program ...

Normal termination,
There were 3 statements executed,
Test case 1 output values.
MAX 2

Is this the correct output? [yn](y) y

After the test case is entered, it is executed on the original program. The tester then verifies that the output result, MAX = 2, is correct. Now the tester can execute the test case against the mutants:
Enter the number of your choice.
1) Enable Mutants
2) Enter Test Case
3) Execute Mutants
4) Status Menu
5) View Live Mutants
6) Test Case Menu
7) View Program Source
8) Execute Mutants in Background
9) Exit CDemo
? 3

Executing testcase #1
16 live mutants.
Executing mutant 1...2...3...4...5...6...7...8...9...10...11...12...13...14...15...1

This test case killed 9 of the 14 mutants, leaving 5 still alive. The 5 remaining mutants represent possible faults in the program and point out inadequacies in the test data. Now the tester can again examine the live mutants to see how to improve the test data.

```
    FUNCTION MAX (M, N)
    MAX = M
    # svr 8     #  MAX = N
    IF (N.GT.M) MAX = N
    # svr 12    #  IF (N.GT. MAX) MAX = N
    # ror 4     #  IF (N.NE. M) MAX = N
    # ror 5     #  IF (N.GE. M) MAX = N
    # ror 6     #  IF (.TRUE.) MAX = N
    RETURN
    END
```

The tester immediately sees that the last mutant shown, number 6, always causes the value for \( N \) to be returned for \( MAX \). The mutant will be easily killed with a test case where \( M \) is the larger value. So he inputs the test case \( M = 2, N = 1 \):
Enter INTEGER value for M: 2  
Enter INTEGER value for N: 1  

Test case #2

Running original program...

Normal termination.  
There were 3 statements executed.  
Test case 2 output values.  
MAX = 2  

Is this the correct output? [yn](y) y

Now the tester executes the live mutants against this new test case:

Executing testcase #2  
5 live mutants.  
Executing mutant 1...2...3...4...5  
End of testcase #2.  2 mutants still alive.

Again, the tester examines the remaining live mutants:

FUNCTION MAX (M, N)  
MAX = M  
IF (N.GT.M) MAX = N  
# svr 12  
# IF (N .GT. MAX) MAX = N  
# rot 5  
# IF (N .GE. M) MAX = N  
RETURN  
END

This time, both mutants left alive are equivalent. Mutant number 12 compares N with MAX instead of with M. At this point in the program, MAX will always be equal to M, so the mutant has no effect. Mutant number five changes the strictly greater than comparison to a greater than or equal comparison. If M and N are equal, then it does not matter which value is assigned to MAX. This mutant will only slow the program down in some cases—it will not change the functional behavior.

With all mutants either killed or shown to be equivalent, the tester can be satisfied that the test cases are adequate relative to the mutants that were generated. Of course for this example only a subset of the mutants were generated. If all types are used, Mothra generates 44 mutants for MAX.

The above example should provide a framework for discussions of both the theoretical and practical aspects of mutation analysis. Although many aspects of the interactions with mutation systems
have changed through the several generations of mutation systems that have been built, the basic iterative cycle of adding test cases, executing mutants, and analyzing the remaining live mutants has been the fundamental way that mutation analysis has been used.

Now that mutation systems have been illustrated, we can describe the major concepts behind the testing technique. A basic assumption behind mutation analysis is called the competent programmer assumption [DeMi78]. This asserts that while programmers do indeed create programs with faults, their programs are "close" to being correct. That is, if asked to write a compiler, the programmers will write a program that is very close to a correct compiler. It may fail on certain inputs, but it will certainly resemble a correct compiler more than, say, an operating system. The mutant operators listed in Table 1 are based on common faults that competent programmers might make.

Another rationale for the mutants that are generated by a mutation system is the coupling effect [DeMi78]. The coupling effect states that test cases that detect simple faults are sensitive enough to also detect more complex faults. In other words, complex faults are coupled to simple faults. The coupling effect cannot be proved, but it has been observed experimentally [Acre79, Acre80]. The coupling effect allows mutants to be simple changes to the program rather than complex changes since test data that kills simple mutants will usually kill more complicated mutants [Budd80].

A program that has been successfully tested against an adequate test set is either correct or contains a fault that has not been modeled by the mutants [DeMi79]. The theoretical properties of adequacy have been discussed by [Budd82]. In practice, a program that has been successfully distinguished from its mutants has been very thoroughly tested [DeMi86, DeMi80].

It is generally impossible to kill all the mutants defined on a program because some changes have no effect. The last mutant shown in MAX is an example of an equivalent mutant that will always produce the same output as the original program and can never be differentiated. Equivalent mutants often represent optimizations or de-optimizations of the original program. In current mutation systems, equivalent mutants are recognized by human examination or by relatively primitive heuristics [Buld79, Tana81]. In fact, a complete solution to the equivalence problem is not possible [Budd82]. The recognition of equivalent mutants and the actual creation of test cases are the two most human-intensive and therefore the most expensive actions within current mutation systems. Reducing the amount of human interaction necessary to perform mutation is a major goal of this work.

Foster describes another error based testing technique that he calls "Error Sensitive Test Case Analysis" (ESTCA) [Fost80]. He presents code errors that are syntactic errors in the source program that will compile correctly. From these are derived three rules for generating test cases that are error sensitive. Unfortunately, he had no means for generating the test cases automatically and so generated them by hand. It is not clear how Foster's work differed from the earlier work on mutation analysis. His code errors are subsets of the mutant operators and the test case generation rules are obvious heuristics that are followed by any user of a mutation tool.

Richardson and Thompson [Rich88] have developed an error based testing technique that is largely derived from path analysis. The RELAY model of error detection defines conditions that guarantee that a fault in a program will originate a failure during execution. A fault originates if some intermediate incorrect state is produced in the program execution. This fault is then guaranteed to transfer through subsequent computations until it is revealed as a failure. The transference, or propagation, of the fault is done mainly through a data flow analysis of the program. Although the research thus far has emphasized the fault transfer and failure detection, the fact that the faults originate for particular classes of faults implies that the RELAY model is an error-based strategy.

The origination conditions of RELAY are similar to the constraints discussed in this dissertation. Rather than being based on mutation operators, they are based on intuitive ideas of how a program may fail—as stated, the focus of the RELAY research has thus far been on the failure propagation.
Morell [More84, More88] has developed a theoretical basis for error-based testing. Part of his work has included a model for contrasting different fault-based techniques. This model is partially based on two concepts: extent, and breadth. A test case causes a local extent effect on the computation if it is able to demonstrate a fault through a local state failure. That is, local extent effects the state of the program immediately after the fault. A global extent effect causes a failure in the final results of the program. The breadth of an error-based technique refers to the size of the fault classes. An infinite breadth technique tests for all possible faults. A finite breadth technique tests for some finite subset of the possible faults.

In an ideal world, we may wish for an infinite breadth, global extent testing technique. Unfortunately, testing for an infinite number of faults does not seem tractable in practice [Budd82]. Because generating test data to ensure global extent testing requires that we cause a failure in the final state of the program, infinite breadth, global extent test data generation is also difficult in practice. So the best we might hope for is a technique that is of finite breadth and local extent. Using these terms, we can contrast the error-based testing techniques available. Figure 2 is a diagram of the two (orthogonal) classifications and where the various techniques lie.

![Figure 2. Error-Based Testing](image)

This diagram illustrates the major differences between the techniques discussed above. The RELAY model is certainly the most ambitious, but undoubtedly it is also impractical. As yet there is no clear model within RELAY of how an infinite number of errors can be detected, and there is no guarantee that failures will propagate to the end of a program execution. Strong mutation analysis is the only technique that fits into the finite breadth/global extent category. Unfortunately, mutation analysis currently has no way to explicitly generate test data. CBT fits into the finite breadth/local extent quadrant. It uses mutation-adequacy to limit the number of errors to a finite set and the extent to a local
measure. This is the principal advantage of constraint based testing—that it is by definition a practical method for generating test data.

Reliability of Test Data

The purpose of the test data selection techniques discussed in the previous section is to choose test data that will allow the tester to infer correctness with a high level of confidence. This section provides a theoretical explanation of why the techniques can be used to raise our confidence in the software. Most of the results surveyed here can be found in [Good75, Howd76, Budd82] and [More88].

For the most part, proofs of theorems are not repeated and the reader is referred to the references for the proofs. The following terms are used throughout this discussion and the dissertation.

A program being tested (including subprograms, subroutines, and functions) is labeled $P$. $P$ is meant to compute a function $F$ with input domain $D$. A finite set of test cases is called $T$, where $T \subseteq D$, and a particular test case is $t \in T$. $P(t)$ is the result of executing $P$ with input $t$.

Given these terms, we now define correctness more formally:

**Definition:** $P$ is correct if:

$$P(t) = F(t) \text{ for all } t \in D$$

An important term used in software testing is that of reliability.

**Definition:** $T$ is a **reliable test set** for $P$ if:

$$P(t) = F(t) \text{ for all } t \in T \implies P(t) = F(t) \text{ for all } t \in D$$

Successful execution of a reliable test set will imply correctness of the program. Conversely, $T$ is reliable for $P$ if $T$ will reveal an fault in $P$, or $P(t) \neq F(t)$ for some $t \in T$. An immediate result is that reliable test sets exist.

**Theorem 1:** $P$ is a program for computing a function $F$ with domain $D$. There exists a finite subset $T$ of $D$ that is reliable for $P$. In other words, there exists a finite set $T \subseteq D$ such that:

$$P(t) = F(t) \text{ for all } t \in T \implies P(t) = F(t) \text{ for all } t \in D.$$ 

This theorem says that reliable test sets exist. In the proof for Theorem 1 [Howd76] chooses a reliable test set based on whether $P$ is correct or not. What is needed is a procedure that will produce reliable test sets. Unfortunately, Howden also proves that no such procedure exists.

**Theorem 2:** There exists no decision procedure that, given an arbitrary program $P$ and function $F$ with domain $D$, can be used to generate a nonempty finite set $T \subseteq D$ such that:

$$P(t) = F(t) \text{ for all } t \in T \implies P(t) = F(t) \text{ for all } t \in D.$$ 

As Weyuker and Ostrand point out [Weyu80], a set of test data is guaranteed to be reliable if and only if it includes the entire input domain. Another strategy is to require that a test set provide some "explanation" of why the program is not incorrect. Test sets that are adequate [DeMi78, Budd82] state that the program is not incorrect and demonstrates that fact with test cases that cause incorrect programs to fail.

**Definition:** $T$ is an **adequate test set** for $P$ that computes $F$ if:

$$P(t) = F(t) \text{ for all } t \in T,$$

and

for all programs $Q$ such that $Q(D) \neq F(D)$, there exists some $t \in T$ such that $Q(t) \neq F(t)$.
In other words, $T$ is adequate for $P$ if the correct version of $P$ behaves correctly on $T$ and all incorrect programs behave incorrectly on at least one element of $T$.

**Theorem 3**: If $T$ is adequate for $P$, then $T$ is reliable, but if $T$ is reliable for $P$, $T$ is not necessarily adequate [Budd80a].

The proof of this theorem uses the fact that adequate test sets do not always exist. This is because an adequate test set must distinguish $P$ from a possibly infinite number of incorrect programs. The following definition is aimed at eliminating this problem by restricting the number of incorrect programs we consider.

**Theorem 4**: Let $A$ be a set of programs. A set of test data $T$ is adequate relative to $A$ if

$$P(t) = F(t) \quad \forall t \in T,$$

for all programs $Q \in A$ if $Q(D) \neq F(D)$, there exists some $t \in T$ such that $Q(t) \neq F(t)$

That is, a set of test data is adequate for $P$ relative to $A$ if the data distinguishes $P$ from all incorrect programs in $A$. The programs in $A$ can be chosen to represent particular faults that the tester decides to test for. Thus, a testing technique using relative adequacy (such as mutation analysis) can require the tester to distinguish between programs that represent specific faults. This allows the tester to test for specific faults.

These results are useful in a variety of ways. Most importantly, they tell us what we can and cannot gain from testing. Specifically, we cannot show correctness through testing. Neither can we generate reliable or adequate test sets. On the other hand, relative adequate test sets do exist, and this dissertation presents a procedure to generate test sets that approximate relative adequacy. A key point to note when comparing reliability with adequacy is the information one gets when no faults are found. When no faults are found using reliability techniques, the tester is left wondering whether the program is correct or if there are faults she has not yet discovered. Adequacy techniques, on the other hand, leave a tester with the knowledge that certain faults are definitely not in the program. In addition, relative adequacy allows us to weaken the concept of adequacy in a way that leaves us with practical ways to measure the quality of the test data. We know of no such theoretic way to weaken reliability.
CHAPTER II
CONSTRAINT BASED TESTING CONCEPTS

Tis but a dream we share, to build
castles out of air.

— Anonymous

This chapter presents the major concepts behind the constraint based testing (CBT) technique. First, the concept of describing adequate test cases with constraints is presented. Then the three major characteristics of a test case that kills a mutant are presented in an informal, intuitive manner. The chapter closes with a restatement of these characteristics and the technique that establishes the method on a theoretical basis.

Generating test data, whether within a mutation system or without, is a difficult task that poses complex problems. Practical test data generation techniques attempt to choose a subset of the possible inputs according to some rationale or testing criteria. The assumption is that this subset of inputs has the properties of being able to find a large portion of the faults in the program as well as being able to establish some confidence in the functional capabilities of the software. A set of test cases that scores well on a mutation system is ensured of having just those properties because the test data is close to being adequate relative to the set of mutant operators. Specifically, the test data will find the set of faults represented by the mutant operators and, if executed successfully, the software is ensured to have none of those faults.

The constraint based test data generation method uses the concepts of mutation analysis to automatically create test data. This test data is designed specifically to kill the set of mutants that are defined on the program. Such test data can be used to kill mutants within a mutation system or used independently as a set of effective test cases. Moreover, since the technique is based on mutation analysis, the test data will be assured of having such properties as containing extremal values [Meyer79], covering all statements [Clar76, Clar85] and all branches [Clar85]. These results are presented in [Acre79]. Studies [Girg86, Walc85] have shown that with proper modification, data that score well on a mutation system can be expected to be more general than any other test data generation method.

The concept of using mutants to generate specific tests was also suggested by Howden [Howd82] in his "weak mutation" approach, by Budd [Budd80a] in his dissertation, and Foster [Fost80] in his error sensitive test case analysis. The present work, however, represents what the author believes is the first general and implemented attempt to automatically generate adequate test sets.

Killing Mutants

In a mutation system, the tester's goal is to create test cases that kill each mutant. Put another way, the tester attempts to select inputs that cause each mutant to fail. For a particular mutant, we say that an effective test case causes the mutant to fail and an ineffective test case does not. In these terms, a test case generator, whether it be a human tester or an automated tool, tries to select effective test cases from the total domain of inputs for the program.

The idea for how the constraint based testing technique generates test data comes from a case study of how testers developed test data when using mutation systems. This study was limited to five testers who were chosen to represent a range of "typical" Mothra users. The most inexperienced tester was an undergraduate who had had no exposure to mutation analysis or to testing. The others were
graduate students who had had some exposure to mutation and testing methodologies. Each of the five testers used the Mothra testing system version 1.0 to develop test cases to kill all the mutants for two programs. In order to simulate typical Mothra users, they were given no guidance other than instruction on how to use the system.

As the five testers added test cases, it became clear that they each followed a similar pattern. Initially they added a few very general test cases and test cases that contained extremal values. This was done by each of the five testers. At some point, each of the testers began analyzing mutants individually and constructing test cases to meet specific requirements. Often, they would jot down little notes of the form "X = Y", or "M > N" when deciding how to kill a particular mutant. These little notes described conditions that they felt were necessary for the test case to kill the mutant. These conditions form the intuitive basis for the constraints that are used in constraint based testing.

An automatic system that generates test cases to kill mutants can be built that follows the same process as a human tester by performing a filtering process. A test case filter will take test cases from the input domain of the program and filter out ineffective test cases, leaving effective test cases. This filter can be described by a set of constraints on a test case, where a test case that satisfies those constraints is effective.

Paths and Constraints

This section describes the characteristics of an effective test case filter. The program BSEARCH is used throughout this section as an example of the concepts. BSEARCH is shown in Figure 3, with one embedded mutant that will also be used for the examples. The mutant is on statement 13 and is a variable replacement mutant (svr) that replaces the variable MID by HIGH.

```plaintext
LOGICAL FUNCTION BSEARCH (LIST, ELEM)
INTEGER LIST (10), ELEM, LOW, HIGH, MID
1 LOW = 1
2 HIGH = 10
3 MID = (LOW + HIGH) / 2
4 IF (HIGH).LT.LOW) THEN
5 BSEARCH = .FALSE.
6 RETURN
7 ELSE
8 IF (ELEM.EQ.LIST(MID)) THEN
9 BSEARCH = .TRUE.
10 RETURN
11 ELSE
12 IF (ELEM.GT.LIST(MID)) THEN
13 LOW = MID + 1
14 ENDIF
15 HIGH = MID + 1
16 ELSE
17 GO TO 10
18 ENDIF
19 ENDIF
20 RETURN
```

Figure 3. Mutant of BSEARCH
To discover how to select a test case that kills an individual mutant, we must first recall that the mutant is represented as a syntactic change to a particular statement. Since each other statement in the mutated program is exactly the same as in the original program, it is apparent that as a minimum we must execute the mutated statement. Assuming the mutated statement is reached, the test case must be such that the state of the mutant program after some execution of the mutated statement differs from the state of the original program after the same execution of the original statement. Although this state difference is necessary, it is not sufficient to kill the mutant. For the mutant to die, the difference in state must propagate through to the end of the program.

For a test case \( T \) to kill a mutant \( M \) appearing on line \( S \) of a program \( P \), \( T \) has to have three broad characteristics. Note that these characteristics make increasingly specific requirements on the test case:

1. Reachability—The statement that contains \( M \) is executed by \( T \).
2. Necessity—The state of \( M \) immediately following some execution of \( S \) is different from the state of \( P \) at the same point.
3. Sufficiency—The final state of \( M \) differs from that of \( P \).

**Reachability**

Reachability refers to the fact that the test case must reach, or execute, the statement that contains the mutant we are trying to kill. To generate test cases that are guaranteed to reach a particular statement \( S \), an expression is needed to describe a path to be taken through the program. This expression is called a *path expression*. A path expression for a statement \( S \) in a program \( P \) is an algebraic expression that describes a condition on a test case that is sufficient to guarantee that \( P \) will reach \( S \). For the mutant of BSEARCH, for example, the test case must reach statement 13, which means the tests in statements 4 and 8 must evaluate to be false and the test in statement 12 must evaluate to be true. The path expression for this mutant would require that \( HIGH \) be greater than or equal to \( LOW \), \( ELEM \) not be equal to \( LIST(MID) \), and \( ELEM \) be greater than \( LIST(MID) \).

**Necessity**

The *state* of a program is described by the value of program variables and internal variables such as the program counter. In order to kill a mutant, a test case must create a state in the mutant program that differs from the state of the original program. Since the mutant program and the original program are identical except for the mutated statement, if the states of the two versions of the program are equal after execution of the mutated statement, then the mutant will not die.

The necessity condition is reminiscent of Howden's weak mutation (Howd82). In weak mutation, the states of the original program and the mutant program are compared over a "component" of the program. The component is a piece of the program that contains the mutated statement. If we view the portion of the program up to and including the mutated statement as a component, then the necessity condition forces the mutated component to produce a different output from the original component. This is exactly the requirement of weak mutation.

In the BSEARCH example, this means that the value of the variable \( LOW \) after the assignment on statement 13 in the mutant program must differ from the value of \( LOW \) in the original program. Since \( LOW \) is assigned the value of \( MID + 1 \) in the original program and \( HIGH + 1 \) in the mutant program,

---

1. In his paper, Howden states that a component will "normally correspond to elementary computational structures in a program". Given a component \( C \) of a program \( P \), there is a mutation transformation that produces \( C' \), and \( P' \) is the mutated version of \( P \) that contains \( C' \).
the necessity constraint for this mutant would require that \( MID \) and \( HIGH \) have different values (\( MID\neq HIGH \)).

The necessary state difference can be described in terms of the same variables and program symbols that the mutant effects. Thus, this change depends solely on the mutant operators. An initial set of necessity constraints are discussed in more detail in Chapter III.

**Sufficiency**

Although a local state change is necessary to kill a mutant, it will not guarantee that the final state of the mutant program will differ from that of the original program. Once the states of the mutant program and the original program diverge, they may well converge to the same, final state. So the constraint is only sufficient to kill a mutant if it ensures that the final state of the mutant will differ from that of the original program.

Richardson and Thompson [Rich88] have been working on this problem. They use an approach based on path analysis. In their terms, a potential fault is a discrepancy between the program being tested and a hypothetically correct version of the program. A potential fault originates if the smallest expression containing the potential fault evaluates incorrectly. The potential fault transfers to a "super"-expression that references the erroneous expression if the value of the "super"-expression is also incorrect. For the potential fault to be detected, the incorrect evaluation must transfer through to the incorrect program. This model of error detection is called the RELAY model because the originations and the successive transfers is analogous to a relay race.

To relate the RELAY model to the constraint based technique, the originations of the potential fault is analogous to the necessity condition to kill a mutant. The transfers of the fault\(^2\) are likewise analogous to the sufficiency condition.

The approach taken in this work is to define a series of increasingly complex approximations to sufficient constraints. The first approximation is to assume that if a test case meets the necessity condition, it will usually meet the sufficiency condition. For the necessity condition but not the sufficiency condition to be met then the mutant does not die and the output of \( M \) is identical to that of \( P \). Since the necessity condition is met, then the state of the mutant program \( M \) does diverge from the original program \( P \) at the point following the mutated statement. That is, the state of \( M \) first diverges from \( P \), then returns to that of \( P \).

This divergence and convergence could happen in one of three ways. First, the test case could be weak—though it had an effect, it did not change the state in a way that would result in a change in the final output. Secondly, the program could be robust enough to recognize the intermediate state as being erroneous and return to a correct state. Thirdly, the intermediate state that the mutant affected could be irrelevant to the final state, in which case the constraint is derived from an equivalent mutant. Although each of these cases is certainly possible, the first seems unlikely—that is, it should occur with relatively low probability. The second case corresponds to fault tolerance in the tested software; the case is interesting enough to warrant further investigation on its own merits. In the absence of fault tolerance however, such robustness is probably rare. The third, of course, provides an opportunity to detect some equivalent mutants, a difficult problem in its own right.

Figure 4 shows an example of another mutant for BSEARCH that illustrates the sufficiency problem. Since statement 3 is always executed, the path expression to statement 3 is NULL. The necessity constraint for the mutant requires the values of \( LOW \) and \( HIGH \) are unequal. So a test case such as \( LIST = (1, 1, 1, 1, 1, 1, 1) \), \( ELEM = 1 \) satisfies the constraints. Since both the

\(^2\) In her paper, Richardson discusses several types of error transfers.
mutant and the original program returns the value of 1 on this test case, it does not kill the mutant. The mutant differs from the original in where the binary search starts looking (at the end rather than the middle), but since all elements of LIST are the same and equal to ELEM, it does not matter where the search starts.

```fortran
LOGICAL FUNCTION BSEARCH (LIST, ELEM)
INTEGER LIST (10), ELEM, LOW, HIGH, MID
1  LOW = 1
2  HIGH = 10
3  10 MID = (LOW + HIGH) / 2
#  10 MID = (HIGH + HIGH) / 2
4  IF (HIGH .LT. LOW) THEN
5     BSEARCH = .FALSE.
6     RETURN
7  ELSE
8     IF (ELEM.EQ.LIST(MID)) THEN
9        BSEARCH = .TRUE.
10     RETURN
11    ELSE
12     IF (ELEM.GT.LIST(MID)) THEN
13        LOW = MID + 1
14    ELSE
15        HIGH = MID - 1
16     ENDIF
17   GOTO 10
18  ENDIF
19 ENDIF
```

Figure 4. Sufficiency Example

**Constraint Based Testing**

This section presents a theoretical basis for the testing technique. Many of the concepts (such as test case adequacy [Good75], the coupling effect [DeMi78], and the competent programmer assumption [DeMi78] ) are shared with mutation analysis and were discussed in Chapter I. This section begins by defining some terms used in testing and stating the characteristics of an effective test case in formal terms, then proceeds to show how to find such test cases given the characteristics. The notations used here, both new and old, are reminiscent of previous software testing work such as [Good75] and [Howd76].

**Definitions**

These definitions are standard and are included here for completeness. A piece of software can be a subroutine, function, a collection of subprograms or an entire program—these are all referred to as programs. The program being tested is labeled \( P \). For each program, there is assumed to exist a (hypothetical) correct version of \( P \), called \( F \). A fault \( b \) in \( P \) will, on at least one input to \( P \), cause \( P \) to return incorrect results. \( b \) is assumed to be located on some particular statement and this statement is referred as \( S_b \). A failure is the run-time erroneous behavior of \( P \). A test case is labeled \( t \), and \( t \) detects \( b \) if the output of \( P \) when executed on \( t \) results in a failure. In this case, \( t \) is effective at finding \( b \), or it demonstrates the fault.
Both $P$ and $F$ are composed of a list of statements. The statements for $P$ are labeled $s_1, s_2, ..., s_N$, and the statements for $F$ are labeled $t_1, t_2, ..., t_M$.

When $P$ is executed on $i$, $i$ causes some control flow $p_1, p_2, ..., p_n$ through $P$, where each $p_i$ is some statement in $P$. Likewise, $i$ causes some (hypothetical) control flow $f_1, f_2, ..., f_m$ through $F$, where each $f_j$ is some statement in $F$.

The program state is the same concept as used in compiler terminology (see, e.g., chapter 2 of Principles of Computer Design [Aho75]) and is taken to be the value of all program data objects, their values, and the program counter. The state of $P$ after execution of statement $p_i$ is called $S(p_i)$. Likewise, the state of $F$ after execution of statement $f_j$ is called $S(f_j)$.

From these definitions, it is clear that $i$ will detect the fault $b$ if and only if the final states differ, that is $S(p_i) \neq S(f_j)$. Of course, in practical situations we are concerned with only a portion of the final state rather than all values.

Fault Detection Conditions

Earlier in this chapter, three broad characteristics of an effective test case were presented. We are now in a position to use the definitions above to state these more precisely. They are stated as conditions that must be satisfied for a test case to detect a fault. These conditions are referred to in the next section and are used throughout the dissertation as the basis for test case generation. The constraints and test cases that are generated from these conditions are analyzed in later chapters. For a test case $i$ to detect a fault $b$, three conditions must be satisfied:

1. The Reachability condition: If $i$ causes some control flow $p_1, p_2, ..., p_n$ to be executed through $P$, then that sequence of statements must include $s_b$.

2. The Necessity condition: $S(s_b) \neq S(t_b)$ on some execution of $s_b$, where $t_b$ is the hypothetical correct version of $s_b$.

3. The Sufficiency condition: $S(p_i) \neq S(f_m)$.

Finding the Fault Detecting Test Case

The above conditions provide a basis for describing effective test cases. In the rest of this section, a theory is developed to formally show that effective test cases can be created. The development is in set theoretic terms and demonstrates both decidable and undecidable aspects of this test data generation method. Although definite results (such as being able to always create effective test cases) are shown to be impossible, probabilistic remarks as to how often effective test cases can be expected to be found are made.

For a program $P$, there is an associated input domain $D$. This domain is represented by the inputs to the program and the range of values such inputs can assume. Each element in $D$ represents a possible test case for $P$. In light of the conditions above, $D$ can be divided in several ways:

- $D = D_R \cup D_N$, where $D_R$ is the portion of the input domain that will reach $s_b$ and $D_N$ is the portion of the input space that will not.

- $D = D_N \cup D_S$, where $D_N$ is the portion of the input domain that will satisfy the necessity condition for $s_b$ and $D_S$ is the portion of the input space that will not.

- $D = D_S \cup D_N$, where $D_S$ is the portion of the input domain that will detect the fault $b$ and $D_N$ is the portion of the input space that will not.

The subscripts $R, N$ and $S$ are taken from the terms reachability, necessity and sufficiency.

If we take the intersection of $D_R$ and $D_N$ ($D_{RN} = D_R \cap D_N$), then the effective test cases for $b$ will lie in this intersection. That is, $D_{RN} \supseteq D_S$. In other words, the test cases that will detect $b$ are a
subset of the test cases that satisfy both the REACHABILITY and NECESSITY conditions.

These sets and their relationships are graphically depicted in the following Venn diagram:

![Venn diagram](image)

Figure 5. Input Domain Subsets

Theorem 5 states that it is undecidable to find a test case that can be guaranteed to reach $s_b$.

**Theorem 5**: Finding $t$ such that $t \in D_E$ is undecidable.

The idea of the proof is that if we are trying to reach the halting statement in a program, then finding a test case that will reach that statement is reducible from the halting problem.

This result tells us that there is no algorithm to find a test case that satisfies the REACHABILITY condition. On the other hand, this condition required that if the test case satisfies the condition then the statement will be executed. This is a very strong requirement. A weaker condition is that if the statement is executed, then the condition will be true.

**Theorem 6**: We can find a predicate such that if a test case $t$ causes $s_b$ to be executed, then $t$ satisfies the predicate.

**Proof**: Define a predicate, WEAK REACHABILITY, to be true. Then any test case that reaches $s_b$ will satisfy WEAK REACHABILITY.

This seems trivial on the surface since any and all test cases will satisfy WEAK REACHABILITY if it is defined to be true. On the other hand, we can certainly use path analysis techniques to construct stronger predicates. As a matter fact, several researchers have used this weaker form of reachability to successfully find test cases that execute statements and/or paths [Clar76, Clar85, Fran86, Chus87]. In Chapter IX we present experiments that show that this predicate can be sufficiently strong to execute a particular statement a large percentage of the time.

We can use the WEAK REACHABILITY approximation to refine the domain subsets shown in Figure 5. If we take $D_E$ to be the portion of the input domain that satisfies WEAK REACHABILITY, then $D_E \subseteq D_b$.

The modified Venn diagram becomes:
Figure 6. Expanded Input Domain Subsets

Just as we defined the subdomain $D_{RN}$ above, we can define the intersection of $D_f$ and $D_h$ to be $D_{RN} = D_f \cap D_h$. So $D_{RN} \supseteq D_h$ and the test cases that detect $b$ are a subset of the test cases that satisfy both WEAK REACHABILITY and NECESSITY.

Remember that our goal is to detect the fault $b$, and the test cases that detect $b$ are contained in $D_h$. Satisfying WEAK REACHABILITY and NECESSITY will give a test case that is in $D_{RN}$, but this does not guarantee that the test case will detect $b$. The question remains, what is the relationship between $D_{RN}$ and $D_h$? If we pick a test case from the region $D_{RN}$, then it may or may not also be in $D_h$. Although the following hypothesis cannot be proved, this chapter concludes with a probabilistic remark to show how, in practical situations, the probability defined can be made to be as high as needed.

**Hypothesis 1**: If $t \in D_{RN}$ then $t \in D_h$ with some probability $p > 0$.

Note that $p$ is not necessarily constant over all programs but depends on the program and the error being detected. Although we will not always pick an effective test case, it is certainly reasonable to pick multiple test cases from the subdomain $D_f \cap D_h$. Given a value for $p$, we can determine how many test cases are needed to reach a certain probability of detecting the program fault.

We can determine how many test cases to pick by asking "if the probability of success is $p$, what is the probability of at least one success over $n$ attempts?". This is equivalent to the probability of not falling on every attempt. Since the probability of failing on $n$ attempts is $(1 - p)^n$, the probability of at least one success over $n$ is given by

$$
\Gamma = 1 - (1-p)^n
$$

This is very promising, since $\Gamma$ grows very quickly when $p$ is positive. If $p$ is zero, of course, then there is no test case that kills the mutant. Values for $\Gamma$ for up to five picks, with several sample values for $p$, are shown below:
This means that even if we have the relatively low probability of finding a fault of .6, we only need to choose four test cases that satisfy both the REACHABILITY and the NECESSITY conditions to have over a 95% chance of detecting the fault. This multiple selection is feasible if we can automate the test case generation technique. An experiment that estimates $p$ to be between .60 and .95 is presented in Chapter IX. Given this result and the multiple selection possibility, we can postulate that a solution to the sufficiency problem (such as that proposed by the RELAY model) is unnecessary.
CHAPTER III
NECESSITY CONSTRAINTS

Start by doing what's necessary, then what's possible and suddenly you are doing the impossible.
— St. Francis of Assisi

The most novel aspect of the technique described in this dissertation is the rationale of constructing the test data to kill mutants. Because the data is designed to kill specific mutants, constraint-based testing will find those types of faults covered by mutation analysis. The basic approach can be stated very simply: for a test case $T$ to kill a mutant, $T$ must make a difference in the mutant's behavior. That is, since the mutant is represented by a single change to the source program, the state of the mutant program must diverge from that of the original program after execution of the mutated statement.

The method used to generate test cases is to describe the test case in terms of arithmetic expressions and other primitive formulae. These descriptions are formalized as constraints on the test case. The constraints are then satisfied to generate input values to the program.

The necessity condition was introduced in Chapter II. To apply this condition to test data generation, we define a set of constraints, called necessity constraints. The necessity constraints are taken directly from the mutant operators defined on a language (such as those in Chapter I, Table 1 for Mothra).

Mutant operators are described in terms of the variables in the program, keywords, and operators. Since the operators are applied to a program as syntactic changes, a test case that meets the necessity condition for a mutant must ensure that the syntactic change results in a state difference for the program. Below, each mutant type for Fortran-77 is described and a corresponding constraint template is developed.

Mutant Operator Constraints

Within a mutation system, the mutant operators define the class of faults being tested for and thusly how close to adequacy the mutation score is. Since the mutants are produced automatically, they are produced by a fixed algorithm. In the Mothra system we defined the construction of mutants as the application of mutant operators to the original program. A mutant operator takes a program as input and produces a set of mutant descriptor records (MDRs). An MDR is a short description of how the original code should be changed to produce the mutant. The MDRs used by Mothra are described in [Offu87] and [DeMi87].

Analogously to mutations, test cases are also produced automatically by a fixed algorithm. To generate test cases, we apply a set of constraint templates to the original program. Each constraint template corresponds to a mutant operator. Rather than producing MDRs, the constraint template takes a program as input and produces necessity constraints that describe the test cases that should kill the corresponding mutant.

The mutant operators, and likewise the constraint templates, can be divided into three groups, depending on whether they affect operands, operators, or whole statements. The next three sections derive constraint templates for the mutant operators in each of these groups.
Source Operand Mutant Operators

The first set of mutant operators alters the basic data objects that the program manipulates. For Fortran 77, there are three types of basic data objects: constants, scalar variables and array references. References to these objects are mutated by replacing a reference to one object with a reference to another:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aar</td>
<td>array for array replacement</td>
</tr>
<tr>
<td>acr</td>
<td>array constant replacement</td>
</tr>
<tr>
<td>asr</td>
<td>array for variable replacement</td>
</tr>
<tr>
<td>car</td>
<td>constant for array replacement</td>
</tr>
<tr>
<td>ccr</td>
<td>constant for scalar replacement</td>
</tr>
<tr>
<td>sar</td>
<td>scalar for array replacement</td>
</tr>
<tr>
<td>scr</td>
<td>scalar for constant replacement</td>
</tr>
<tr>
<td>src</td>
<td>source constant replacement</td>
</tr>
<tr>
<td>svr</td>
<td>scalar variable replacement</td>
</tr>
</tbody>
</table>

Table 3. Replacement Mutants

These mutants yield the easiest to an example, the first mutant shown in the MAX program in Chapter 1 was an aar mutant where \( M \) was replaced by \( N \). To kill that mutant, a test case must have different a value for \( M \) than for \( N \). So the constraint \( M \neq N \) is needed. This constraint is generalized to the form \( X \neq Y \), where \( X \) and \( Y \) represent arbitrary scalar variables. Likewise, each of the nine mutant types in Table 3 yields a constraint template in which the two data objects have unequal values. All the replacement mutant types with their corresponding constraint templates are in Table 4. In the table, the symbols \( X \) and \( Y \) are used for scalar variables, \( A \) and \( B \) are used for array names, \( e_1 \) and \( e_2 \) are used for arbitrary expressions, and \( M \) and \( N \) are used to represent constants. This notation will be used throughout the remainder of this chapter.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Mutant</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>aar</td>
<td>array for array replacement</td>
<td>( A(e_1) \Rightarrow B(e_2) )</td>
<td>( A(e_1) \neq B(e_2) )</td>
</tr>
<tr>
<td>acr</td>
<td>array constant replacement</td>
<td>( M \Rightarrow A(e_1) )</td>
<td>( M \neq A(e_1) )</td>
</tr>
<tr>
<td>asr</td>
<td>array for variable replacement</td>
<td>( X \Rightarrow A(e_1) )</td>
<td>( X \neq A(e_1) )</td>
</tr>
<tr>
<td>car</td>
<td>constant for array replacement</td>
<td>( A(e_1) \Rightarrow M )</td>
<td>( A(e_1) \neq M )</td>
</tr>
<tr>
<td>ccr</td>
<td>constant for scalar replacement</td>
<td>( X \Rightarrow M )</td>
<td>( X \neq M )</td>
</tr>
<tr>
<td>sar</td>
<td>scalar for array replacement</td>
<td>( A(e_1) \Rightarrow X )</td>
<td>( A(e_1) \neq X )</td>
</tr>
<tr>
<td>scr</td>
<td>scalar for constant replacement</td>
<td>( M \Rightarrow X )</td>
<td>( M \neq X )</td>
</tr>
<tr>
<td>src</td>
<td>source constant replacement</td>
<td>( M \Rightarrow N )</td>
<td>( M \neq N )</td>
</tr>
<tr>
<td>svr</td>
<td>scalar variable replacement</td>
<td>( X \Rightarrow Y )</td>
<td>( X \neq Y )</td>
</tr>
</tbody>
</table>

Table 4. Constraint Templates—Replacement Mutants

Another type of source operand mutation works with arrays of the same dimensionality. For comparable name replacements, only the array name is replaced, not the full array reference (as in the aar mutant). Again, a test case to kill that mutant needs to have different values for the two arrays at that index.

Another source operand mutation operator takes each constant in the the program and "alters" it slightly. These constant replacement mutants increment and decrement each integer constant by 1, each real constant by 10%, and take the complement of each logical constant. Also, the first character in each string constant is replaced by its predecessor and successor in the underlying collating sequence. (For example, 'MUTATION' is replaced by 'LUTATION' and 'NUTATION'.) The DATA statement
alteration mutant performs the same alterations to constants in DATA statements.

These three mutations, along with their corresponding constraint templates, are shown in Table 5.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Mutant</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>crp</td>
<td>constant replacement</td>
<td>( M \Rightarrow N )</td>
<td>( M \neq N )</td>
</tr>
<tr>
<td>daa</td>
<td>data statement alteration</td>
<td>( M \Rightarrow N )</td>
<td>( M \neq N )</td>
</tr>
<tr>
<td>cmr</td>
<td>comparable array replacement</td>
<td>( A(e_1) \Rightarrow B(e_1) )</td>
<td>( A(e_1) \neq B(e_1) )</td>
</tr>
</tbody>
</table>

Table 5. Constraint Templates—Constraint Replacement Mutants

Operator Mutations

Operator mutations perform mutations on operators of the language. There are five types of operator mutations, including several special operators that are added to the Fortran 77 set of operators to enable mutation to test for certain types of faults. These new operators are produced only in mutated programs.

The first two new operators are binary operators that can replace either arithmetic or logical operators. The LEFTOP operator returns the value of the left operand and RIGHTOP returns the value of the right operand. For logical operators or relational operators, the two binary operators TRUEOP and FALSEOP have been added to return either TRUE or FALSE, regardless of the operands.

These operators are used in arithmetic operator replacement mutations, logical operator replacement mutations, and relational operator replacement mutations. Arithmetic operator mutations replace each occurrence of an arithmetic operator with every other arithmetic operator that is legal at that point (as well as the new operators mentioned above). The logical and relational operator replacements perform the same actions on their respective operators.

To kill operator mutations, we require that the expressions involving the mutants produce different results from the expression in the original program. Table 6 summarizes these three mutant operators with the corresponding constraint templates. In the table, \( \gamma \) and \( \lambda \) are used to represent binary operators.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Mutant</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>aor</td>
<td>arithmetic operator replacement</td>
<td>( e_1 \gamma e_2 \Rightarrow e_1 \lambda e_2 )</td>
<td>( e_1 \gamma e_2 \neq e_1 \lambda e_2 )</td>
</tr>
<tr>
<td>(LEFTOP)</td>
<td></td>
<td>( e_1 \gamma e_2 \Rightarrow e_1 )</td>
<td>( e_1 \gamma e_2 \neq e_1 )</td>
</tr>
<tr>
<td>(RIGHTOP)</td>
<td></td>
<td>( e_1 \gamma e_2 \Rightarrow e_2 )</td>
<td>( e_1 \gamma e_2 \neq e_2 )</td>
</tr>
<tr>
<td>(MOD)</td>
<td></td>
<td>( e_1 \gamma e_2 \Rightarrow \text{Mod}(e_1,e_2) )</td>
<td>( e_1 \gamma e_2 \neq \text{Mod}(e_1,e_2) )</td>
</tr>
<tr>
<td>lcr</td>
<td>logical connector replacement</td>
<td>( e_1 \gamma e_2 \Rightarrow e_1 \lambda e_2 )</td>
<td>( e_1 \gamma e_2 \neq e_1 \lambda e_2 )</td>
</tr>
<tr>
<td>mcr</td>
<td>relational operator replacement</td>
<td>( e_1 \gamma e_2 \Rightarrow e_1 \lambda e_2 )</td>
<td>( e_1 \gamma e_2 \neq e_1 \lambda e_2 )</td>
</tr>
</tbody>
</table>

Table 6. Constraint Templates—Operator Mutants

There are also two kinds of mutations that insert operators. Absolute value operators insert one of three operators before arbitrary expressions; ABS, NEGABS, and ZPUSH. The ABS operator is a Fortran 77 operator, but the NEGABS and ZPUSH were added for mutation analysis. The NEGABS operator takes the negation of the absolute value of its argument and the ZPUSH operator raises an exception (killing the mutant) only if its argument is zero. These three mutants are killed by requiring the expression to take on a negative value, a positive value, and a zero value.
Unary operator insertions negate each numeric expression, increment each numeric expression by 1, and decrement each numeric expression by 1. In addition, each logical expression is complemented. The operator insertion mutations and constraint templates are shown in Table 7.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Mutant</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>absolute value insertion</td>
<td>$e_1 \Rightarrow \text{ABS}(e_1)$</td>
<td>$e_1 &lt; 0$</td>
</tr>
<tr>
<td></td>
<td>(NAGABS)</td>
<td>$e_1 \Rightarrow \text{NEGABS}(e_1)$</td>
<td>$e_1 &gt; 0$</td>
</tr>
<tr>
<td></td>
<td>(ZPUSH)</td>
<td>$e_1 \Rightarrow \text{ZPUSH}(e_1)$</td>
<td>$e_1 = 0$</td>
</tr>
<tr>
<td>uni</td>
<td>unary value insertion</td>
<td>$e_1 \Rightarrow e_2$</td>
<td>$e_1 \neq e_2$</td>
</tr>
</tbody>
</table>

**Table 7. Constraint Templates—Insertion Mutants**

**Statement Mutations**

The last kind of mutant operator includes five distinct mutations that perform changes to entire statements. The DO-statement-end replacement operator replaces the target label of a DO loop with every other label in the same program unit. In addition, each DO statement is replaced by a ONETRIP statement that causes the loop body to always be executed at least once. For the label replacement, we require that the loop be executed at least twice. So if the statement is of the form "DO L. $x = e_1, e_2$", we require that the value of $e_2$ be at least 2 greater than $e_1$. For the ONETRIP mutant, we require that the original loop not be executed at all, so $e_2$ must be less than $e_1$.

The other four mutations replace each occurrence of a label of a GOTO statement with every other label in the program unit and replace every statement by a RETURN statement, a CONTINUE statement, and a TRAP statement. A TRAP statement causes a failure, ensuring that each statement in the program is executed. The necessity constraints for these mutants are NULL because the mutant operators are defined in such a way that the mutant state is always different from that of the program state. In particular, these mutants change the flow of control so the mutant always changes the program counter. So the only requirement is that statement with the mutant is reached. Table 8 summarizes the statement mutations and the constraint templates.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Mutant</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>der</td>
<td>DO statement end replacement</td>
<td>$DO L_1 X = e_1, e_2 \Rightarrow DO L_2 X = e_1, e_2$</td>
<td>$e_2 - e_1 \geq 2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$e_2 &lt; e_1$</td>
</tr>
<tr>
<td>gtr</td>
<td>goto label replacement</td>
<td>GOTO L_1 \Rightarrow GOTO L_2</td>
<td>NULL</td>
</tr>
<tr>
<td>rar</td>
<td>return statement replacement</td>
<td>S \Rightarrow RETURN</td>
<td>NULL</td>
</tr>
<tr>
<td>san</td>
<td>statement analysis</td>
<td>S \Rightarrow CONTINUE</td>
<td>NULL</td>
</tr>
<tr>
<td>san</td>
<td>statement deletion</td>
<td>S \Rightarrow TRAP</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Table 8. Constraint Templates—Statement Mutants**

1. In Fortran 77, if the initial index value of the loop is greater than the final index value of the loop, the loop is not executed.
Predicate Constraints

Early experimentation with a prototype implementation of this testing technique revealed a major inadequacy with the necessity constraints as initially defined. The problem was with killing mutations that were in predicate expressions. Although the test cases being generated would cause an immediate effect on the state of the mutant program, the (binary) result of the mutant predicate often would be the same as of the original program. In this section, this problem is presented and a solution is developed. This solution has been implemented in the Godzilla generator. An example of the inadequacy of the necessity constraints is shown in Figure 7.

<table>
<thead>
<tr>
<th>Mutant</th>
<th>IF (I+K.GE.J) THEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Necessity constraint</td>
<td>i ≠ 3</td>
</tr>
<tr>
<td>Test case</td>
<td>I = 7, J = 9, K = 7</td>
</tr>
</tbody>
</table>

Figure 7. Predicate Problem

Figure 7 shows a constant for variable replacement where the constant "3" is used instead of the variable "I". Although the test case satisfies the necessity constraint (as shown), the result of the predicate test remains unchanged. $I+K = 14$ and $3+K = 10$, both of which are greater than $J$.

This problem suggests an extension to the necessity constraints that force mutations of predicates to have different results than the corresponding predicate in the original program. The technique to ensure a predicate difference is as follows. For each mutation on an expression $e$ in the program, construct the mutated expression $e'$. From these, construct the predicate constraint as $e \neq e'$. In the example above, the predicate constraint would be:

$$(I+K > J) \neq (3+K > J)$$

The test case in Figure 7 would substitute into this predicate constraint as $((7+7 > 9) \neq (3+7 > 9)) = FALSE$. A test case to satisfy this constraint might be $7, 10, 7$, in which case $I+K = 14$ and $3+K = 10$. This would make $((7+7 > 10) \neq (3+7 > 10)) = TRUE$.

The fact that the constraint-based technique is robust enough to allow constraints such as this to be added is intriguing. Perhaps other types of constraints could be added to further refine the constraints or to add more error-detection power to the test cases. For example, it could well be possible to include domain testing [Whit80, Hass80] or other path-based testing techniques as additional constraints.
CHAPTER IV

COMPUTING PATH EXPRESSIONS

Path expressions were introduced in Chapter II as an expression that describes a path through a program $P$ to a statement $S$. In this chapter, path expressions are described in more detail and an algorithm for computing them is presented.

Since a mutation is represented by a change to a particular statement, that statement must be executed in order to distinguish the mutant. Not only must the test case satisfy the necessity constraint, it must also reach $S$. This can be accomplished by constructing a path expression that describes possible paths to each statement in the program. Ideally, we would like to construct a path expression that describes all paths to a statement. In Chapter I, we pointed out that this cannot, in general, be done. In practice, we can construct a path expression that approximates all paths. In subsequent discussions, these will be distinguished as complete path expressions and partial path expressions where complete path expressions describe all paths to a statement and partial path expressions describe some of the possible paths. The automatically constructed path expression, $PE$, is assumed to be a partial path expression and is conjoined with the necessity constraint to create a more effective filter for the test cases.

In Godzilla, the partial path expression for each statement in a program unit is constructed by a symbolic walk-through of the program [King76, Howd77, Howd78]. Each statement is examined in turn and the current $PE$ is updated according to the type of statement. $PE$ is initially given the default value TRUE; and each statement is given the value FALSE, indicating that no path to that statement has been found. Upon reaching a statement $S$, several actions are performed. First, the current $PE$ is added to the path expression list for $S$. $PE$ represents a new way of reaching $S$, and each possible path to $S$ is stored as a separate path expression. Next, all $PE$'s previously stored for $S$ are disjointed to the current $PE$. This represents several ways of getting to the statement following $S$. Finally, if $S$ is a control flow statement, the current $PE$ is updated by a modification rule that depends on which type of statement $S$ is.

For example, suppose that $S$ is a GOTO statement. The current $PE$ is added to the path expression list of the statement that is the target of the GOTO. Next, the current $PE$ is given the value of FALSE, because the statement following $S$ cannot be reached through $S$—control always transfers.

Table 9 shows the $PE$ modification rule for each type of Fortran 77 control flow statement. These rules are based on input and output predicates as used in program verification. They are modified from predicate transformation rules for generating verification conditions (see, e.g., chapter 3 of Mathematical Theory of Computation [Mann74]). The rules shown in Table 8 are applied during the path expression algorithm to modify the current $PE$.
<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Path Expression Rule</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Statement</td>
<td>( PE = TRUE )</td>
<td></td>
</tr>
<tr>
<td>IF (e 1) THEN</td>
<td>( PE )</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>( PE = PE \land \neg e 1 )</td>
<td></td>
</tr>
<tr>
<td>ELSE IF (e 2) THEN</td>
<td>( PE = PE \land \neg e 1 \land e 2 )</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>( PE = PE \land \neg e 1 \land \neg e 2 )</td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>( PE = PE \land \neg e 1 \land \neg e 2 )</td>
<td></td>
</tr>
<tr>
<td>ENDIF</td>
<td>( PE )</td>
<td></td>
</tr>
<tr>
<td>IF (e) S</td>
<td>( PE = PE \land e )</td>
<td>Applies to S, not IF.</td>
</tr>
<tr>
<td>IF (e) L1L2L3</td>
<td>( PE )</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>( \text{FALSE} )</td>
<td></td>
</tr>
<tr>
<td>L1 S2</td>
<td>( PE = PE \land e &lt; 0 )</td>
<td></td>
</tr>
<tr>
<td>L2 S3</td>
<td>( PE = PE \land e = 0 )</td>
<td></td>
</tr>
<tr>
<td>L3 S4</td>
<td>( PE = PE \land e &gt; 0 )</td>
<td></td>
</tr>
<tr>
<td>DO L1 = a,b,c</td>
<td>( PE )</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>( PE = PE \land a \leq b )</td>
<td>if ( c \geq 0 )</td>
</tr>
<tr>
<td>L S2</td>
<td>( PE )</td>
<td></td>
</tr>
<tr>
<td>DO L1 = a,b,c</td>
<td>( PE )</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>( PE = PE \land a \geq b )</td>
<td>if ( c \leq 0 )</td>
</tr>
<tr>
<td>L S2</td>
<td>( PE )</td>
<td></td>
</tr>
<tr>
<td>All others</td>
<td>No change</td>
<td></td>
</tr>
</tbody>
</table>

Table 9. Path Expression Rules

We can now give a complete description of the partial path expression construction algorithm used in Godzilla:
Algorithm 1: COMPUTING PATH EXPRESSIONS

Variables: 
- \( CPE \) is the current path expression.
- \( PE \) \( (\cdot) \) contains the current path expression for each statement.
- \( P \) is the program.
- \( s \) is a statement in program \( P \).

1. [Initialize]
   \[
   PE = \text{TRUE} \\
   \text{for each statement } s \text{ in } P \\
   \quad PE \ (s) = \text{FALSE}
   \]
2. for each statement \( s \) in \( P \) in order
3. \[
   PE \ (s) = PE \ (s) \lor CPE
   \]
4. \[
   CPE = PE \ (s)
   \]
5. if \( s \) is a control flow statement then
6.   Update \( CPE \) according to Table 9
7. end

Figure 8. Path Expression Algorithm

When the algorithm terminates at line 7, \( PE \ (s) \) will be satisfied if \( s \) is executed. If there are \( N \) statements in \( P \), then each of the two loops will execute \( N \) times. The assignments in line 1 and on lines 3 and 4 are constant time. The update on line 5 is a table lookup and assignment, so it can be done in constant time. Thus, the algorithm will run in time proportional to \( N \), so the algorithm is linear in the number of statements.

Since we know that Algorithm 1 will not construct all paths in every statement, a natural question is "what types of paths will it fail to find?". Algorithm 1 is a one-pass algorithm. It examines each statement once. When it reaches forward branches, it computes and saves the path expression for the forwardly referenced statement. When the algorithm reaches that statement, it will use the path expression calculated from the forward branch in subsequent statements. When Algorithm 1 reaches back-branches, it also computes and saves the path expression for the statement that is branched to. Algorithm 1 does not return to the statement that was reached via a back-branch and propagate this new path expression through the subsequent statements. These are precisely the types of paths that the path expression algorithm used by Godzilla fails to find.

One construct that use back-branches implicitly is that of loops (in Fortran, DO loops). Each iteration through a loop takes an implied back-branch. Because Algorithm 1 computes the conditions to enter the loop, it will find at least one path (if any are available) to reach the statements within the loop. Explicit back-branches (GOTO statements) can cause a more serious problem for this algorithm. In pathological cases, Algorithm 1 might find no paths to a statement when that statement can only be reached through back branching. For example consider the following partial code fragment:
Since the statement with label 2 can only be reached through the last GOTO statement, Algorithm 1 will not produce any path expression to the statements between this statement and the statement with label 1. Although this is certainly a contrived, pathological case, this situation could arise in practice. An interesting point is that most adherents to structured design and coding practices would suggest not using back-branches in a program. This problem with the path expression algorithm suggests an analytical reason for writing structured programs because the programs would be easier to develop path expressions for and easier to test.

One way of combating the back-branching problem is to traverse the program twice rather than once as in the above example. This should reduce the number of statements for which no path expression has been generated, while only increasing the complexity of the algorithm by a constant factor. We could still construct programs for which the algorithm would not produce any paths to a statement that is, in fact, reachable, but intuitively at least we would expect that these programs would be even less likely to occur in practice. If we continue to increase the number of times we execute the path expression algorithm, we will eventually reach the point of diminishing returns where we are spending a great amount of time computing path expressions that may not significantly increase the adequacy of the test data.

It is not clear that this back-branching problem is significant. Of the five Fortran programs studied in the experiments described in Chapter IX, several of the experimental programs used DO-loops and two of the programs (FIND and GCD) contained back-branches. Of the 3600 constraints generated for these five programs, there were a total of 654 equivalent mutants. There was not a single infeasible constraint system that did not represent equivalent mutants.

The path expressions for BSEARCH are shown in Figure 10. Note that the path expression for statement 1 is TRUE. This is because statement 1 will always be executed.
Figure 10. BSEARCH Path Expressions
CHAPTER V
CONSTRAINT SATISFACTION

I can't get no satisfaction!

— Mick Jagger

In previous chapters, the idea of using constraints to specify test cases was presented. These constraints are mathematical formulas that involve program variables and operators of the language. Test cases are created by choosing values that satisfy the constraints. In this chapter, the terms "constraint" and "satisfaction" are defined precisely and several procedures for performing constraint satisfaction are presented.

Some of the satisfaction procedures are drawn from Gosling's dissertation [Gosl83], which was entirely on the subject of constraints. After this review, a procedure for solving the constraints of constraint-based testing is developed that take advantage of certain simplifications inherent in the test case constraints. This new procedure is adapted from previous procedures. To satisfy test case constraints, we do not attempt to find an exact solution to the sufficiency problem. Software testing is an imperfect science and we see no reason for satisfaction procedures to be perfect. Rather, a constraint satisfaction procedure must be effective and must never give a wrong answer—a test case that does not satisfy the constraints.

Experiments that analyze the efficiency and effectiveness of the procedures are presented in Chapter IX.

Definitions

The term "constraint" is used widely, in problems ranging from computability theory [Hopc79] to artificial intelligence [Rich83]. The definitions presented here are based on those used in linear programming (see, e.g., chapter 1 of Linear Programming and Network Flows [Baua77]).

Definition: An algebraic expression is an expression composed of variables, parentheses, and programming language operators.

The variables are symbols found in some program, and the programming language operators are any of the operators that the programming language being considered defines over the variables in the constraint. The variables and the algebraic expression can take on any value that the programming language allows.

Definition: A constraint is a pair of algebraic expressions related by a conditional.

A conditional is one of the operators \( \geq, <, =, \geq, \leq, \neq \). If \( A_1 \) and \( A_2 \) are algebraic expressions, then \( A_1 \geq A_2 \) is a constraint. A constraint can take on one of the binary values \text{true} or \text{false}. Constraints can also be modified by the negation operator \( \text{not} \).

Definition: A constraint system is a sequence of constraints connected by the two logical operators \( \land \) and \( \lor \).

A constraint system can take on either the value \text{true} or \text{false}. Note that logical connectors are often included in the programming language operators. A constraint system is often referred to as "constraints" or even just "constraints" when the distinction is obvious or unimportant.
Definition: A clause is a set of constraints connected by only one type of the two logical operators. A disjunctive clause uses only the logical OR and a conjunctive clause uses only the logical AND.

Definition: A constraint system is in disjunctive normal form if it is in the form $E_1 \lor E_2 \lor \cdots \lor E_k$, where each $E_i$ is a conjunctive clause. A constraint system is in conjunctive normal form if it is in the form $E_1 \land E_2 \land \cdots \land E_k$, where each $E_i$ is a disjunctive clause.

Constraints that are in either disjunctive or conjunctive normal form (DNF or CNF, respectively) are generally easier to manipulate with software. The software knows the form of the constraints. The Godzila implementation, for example, expects and keeps all constraints in disjunctive normal form.

Given these definitions, we can now describe constraint satisfaction. Generally, the vector of values that satisfy the constraint system are assumed to be in "positional" order, that is, the first value in the vector is assigned to the first variable in the constraint system, and so on. This assignment can also be made explicit by pairing values with variables, i.e., $x_i = v_j$.

Definition: A vector of values $v_1, v_2, \ldots, v_k$ is called a satisfying point for a constraint system $C$ if the values can be assigned to the variables in $C$ such that $C$ evaluates to TRUE.

In the following definition, the term "connected" is used in an intuitive sense. We assume that two points are connected if they are adjacent in the discrete number space. I.e., if in a 2 dimensional space, two points are adjacent if they lie next two each other on the line.

Definition: A satisfying region for a constraint system is a set of connected satisfying points for a constraint system.

Definition: The satisfying space for a constraint system is the set of all satisfying points for a constraint system.

Note that there may be multiple, unconnected, satisfying regions in the satisfying space. When there is only one region in the satisfying space, "region" is sometimes used informally to indicate the entire space.

Definition: Two systems of constraints that have the same satisfying space are equivalent.

Informally speaking, if the constraints are modeling some situation or object, they are said to have a semantic meaning. So two equivalent constraint systems are sometimes called semantically equivalent.

Definition: A constraint system is feasible if there is at least one satisfying point. A constraint system is infeasible if there is no satisfying point.

With these definitions, we can now define the constraint satisfaction problem. Given a constraint system $C$, the satisfaction problem for $C$ is the problem of finding at least one satisfying point.

For the constraint based method, each necessity constraint/path expression pair forms a constraint system. A satisfying point corresponds to a test case. The process of choosing a value for a test case variable is called assigning the variable or fixing the variable. The process of checking to see if a constraint system is satisfied by a test case is called evaluation. In the satisfaction procedures presented below, an procedure to evaluate the system of constraints on a point is often assumed.

**Constraint Satisfaction Methods**

In this section, we review a general method for solving systems of constraints called propagation. The method does not always succeed, and two ways of handling situations in which it fails are presented. The discussion develops a background for methods that are more specific for solving test case constraints,
which are presented in the next section.

Propagation

Various forms of the propagation procedure have been used in many constraint systems [Gosl83, Stee80, Born79]. The procedure is based on the topological sort algorithm [Hara69], is simple to understand and runs in polynomial time [Gosl83]. Unfortunately, propagation fails in many situations. Simply put, propagation attempts to assign values to variables using only local information. When a variable is given a value, this value is then substituted back into the other constraints until all variables can be assigned a value. Propagation substitutes values and computes expressions until either all values have been deduced, an inconsistency has been encountered, or no more deductions can be made.

Below is a description of the propagation procedure. When the procedure starts, some variables may have values that are known ahead of time. For example, constraints such as $X = K$, where $K$ is a constant give us a known value for $X$. During execution of the procedure, variables can be assigned a value in one of two ways. The constraint system may be simplified to where there is only one possible value. For example, the constraint system may be simplified to a point where it contains $X > 0$ and $X < 2$. In this case, $X$ must equal 1. The other way the procedure may assign a value to a variable is by pseudo-randomness. For example, the constraint system may be simplified to a point where it contains $X > 0$ and $X < 10$. In this case, $X$ can be assigned a value between 0 and 10 and a value can be chosen arbitrarily from this domain.
Algorithm 2: PROPAGATION

Variables: \( Next \) is a set of constraints.
\( TC \) is an array of values, indexed by the variables in the constraints.
\( C \) is the system of constraints.
e is a constraint clause.
x is a variable.
v is a value.

1. [Initialize]
   for each constraint e in C
     if there is a variable x found in e whose value is known then
       Place e in Next, remove e from C
   end
2. [Initialize]
   for each variable x in C
     if there is a known value v for x then
       \( TC \{ x \} = v \)
     else
       \( TC \{ x \} = \text{undefined} \)
   end
3. while Next is not empty
4.   Choose (and remove) some constraint e from Next
5.   for each variable x in e
6.     if \( TC \{ x \} \neq \text{undefined} \) then
7.       replace the variable x in e with \( TC \{ x \} \)
8.   end
9.   if all variables x in e are assigned then
10.      if e = TRUE then
11.        remove e from further consideration
12.      else
13.        C cannot be solved. return INFEASIBLE.
14.   else if there is only one unassigned variable x in e
15.     Use the other variables in e to deduce a value v for x
16.     \( TC \{ x \} = v \)
17.     for every constraint e' in C
18.       if x is in e' then
19.         Remove e' from C, add e' to Next
20.     end
21.     else
22.       replace e in C
23.     end
24.   if C is not empty then
25.     propagation fails. return INFEASIBLE
26.   else
27.   return FEASIBLE

Figure 11. Propagation Procedure
If the procedure ends at line 27, then the values in \( TC \{x\} \) represent a solution to \( C \). If the procedure ends at line 13 or 25, then the constraints cannot be solved by propagation. Let \( N \) be the number of constraints in \( C \). Let \( M \) be the number of values required for \( TC \) (the number of variables). When a constraint \( c \) is removed from \( \text{Next} \), one of two things can happen to it. If all the variables in \( c \) are fixed (line 9) or become fixed in line 15, then \( c \) is removed from consideration. If not, then \( c \) is placed into \( \text{C} \) to be evaluated later. Constraints in \( \text{C} \) can only be moved into \( \text{Next} \) (in line 19). So as the procedure works, constraints are cycled back and forth between \( \text{C} \) and \( \text{Next} \) until the procedure terminates or until the constraint system is satisfied. Because each time a constraint is taken from \( \text{Next} \), one variable will be fixed and it can be taken from \( \text{Next} \) at most \( M \) times. If there are \( N \) constraints, the loop starting at line 3 will run \( N \times M \) times. There is an inner loop at line 17. It executes at most \( N \) times. So the propagation procedure runs in time at most \( M^N \).

As an example of constraint satisfaction using propagation, consider the following two constraints:

**Example 1:**

\[
\begin{align*}
Z &= Y + A \\
Y &= X + A
\end{align*}
\]

They constrain \( Y \) to be the average of \( X \) and \( Z \). Suppose we know that \( X = 3 \) and \( Y = 4 \). Then \( A \) can be deduced from

\[
A = Y - X = 4 - 3 = 1.
\]

From this, \( Z \) is evaluated as:

\[
Z = Y + A = 4 + 1 = 5.
\]

On the other hand, suppose we know that \( X = 2 \) and \( Z = 4 \). This tells us nothing about the other values since each constraint only has one known value out of three. The equations become:

\[
\begin{align*}
4 &= Y + A \\
Y &= 2 + A
\end{align*}
\]

Of course, we could use algebraic manipulations to solve for \( A \), yielding \( A = 4 - Y \). This can be substituted into the second equation, giving \( Y = 2 + 4 - Y \), or \( Y = 6/2 \). Unfortunately, this type of algebraic manipulation is very difficult, and the propagation procedure as such does not have any means of doing it.

This type of algebraic expression is called a cycle because of the a circular dependency among the variables. The term cycle comes from the dependency graph on the equations. The dependency graph is formed as follows. Each variable in the equations is given a node. For each occurrence of a variable in the right hand side of the equation, an undirected edge is drawn from its node to the node of the variable on the left hand side. Given this graph, a cycle is defined as in standard graph theory (see, e.g., *Graph Theory* [Har69]) to be a path through the graph that starts and finishes at the same node. The dependency graph for Example 1 is shown in Figure 12. Assigning values to \( X \) and \( Y \) allow us to break the cycle between \( A \), \( Y \), and \( X \), whereas assigning to \( X \) and \( Z \) does not.
Figure 12. Dependency Graph

Much of the complexity of constraint satisfaction is due to cycles, and we will mention cycles throughout this chapter. Because propagation uses only local information to satisfy constraints, it can be called a greedy procedure. In the presence of cycles, this local information is not always enough to satisfy the constraints, and local propagation fails. Since one of the hardest problems with satisfying constraints is due to cycles in the expressions, it is not surprising that methods exist that specifically try to handle cycles. The next two techniques described were designed specifically to handle cycles in constraints.

Relaxation

Relaxation is a method used often in numerical approximation problems [Bun78]. It is a general technique for iteratively finding solutions to systems of equations. In a relaxation procedure, each variable is assigned an initial value. Then an error estimate is computed for each equation and a total system error estimate is computed from the sum of the errors for each equation. Relaxation attempts to minimize the system error estimate by iteratively modifying the values in the equations.

The calculation of the error estimate is highly dependent on the problem being solved. Error estimates are most accurate when each equation is an equality where the right-hand-side of the equation is a constant. Although arbitrary constraints can usually be transformed into this form, converting an inequality to an equality generally reduces the solution space for the constraint by a dimension (from a half space to a line, for example).

Relaxation procedures are not guaranteed to converge to a zero error estimate. Whether they do converge depends largely on the initial values, how the error estimates are computed, and how the values are modified. One problem is that even though the error estimates may give a good approximation of how bad the current values are, they do not indicate how to improve the error estimates. For most linear systems of equations, relaxation does converge; unfortunately it usually converges very slowly. For these reasons, relaxation is not a good technique for satisfying test case constraints.

Redundant Constraints

Borning [Bor79] and Steele [Ste80] independently suggested the idea of breaking cycles by adding redundant constraints to the cyclic equations. These additional constraints can allow enough constraints to be present so that the equations can be solved by propagation. Returning to Example 1, we can add two constraints as follows:

\[
\begin{align*}
Z &= Y + A \\
Y &= X + A \\
Z &= X + B \\
B &= A + A
\end{align*}
\]
These two equations introduce an artificial variable \( B \). With these equations, if we are given values for any two of \( X, Y, \) or \( Z \), the other variable can be determined using propagation.

A system of constraints can be satisfied by any point in the satisfying space. When adding redundant constraints, the new constraints must not change this space. That is, the new system of equations must still be satisfied by the same set of solutions (plus values for artificial, arbitrary variables). There are few rules for adding constraints to an already existing system of equations,\(^1\) so systems that use this method require the user to add redundant constraints by hand. This has the potential for introducing errors into the system by allowing the user to accidentally change the satisfying space for the constraint system. Moreover, the interaction can be intense and frustrating because the user will not always be sure that the added constraints will allow a solution to be found.

**Satisfaction of Test Case Constraints**

The above discussion should make it clear that propagation procedures are simple and efficient. Unfortunately propagation does not always work—largely because of the problems with cycles in the constraints. This section discusses ways to satisfy constraints within a test case generation system. A series of procedures is presented that range from simple and slow to complicated and efficient.

**Random Satisfaction**

The most obvious method for satisfying constraints is to choose random values for each variable involved in the constraints until the constraints are satisfied. This procedure has the obvious advantage of being simple and quick to implement. It has the obvious disadvantages that it might be slow find a satisfying solution and may not even find a solution if one exists.

---

1. Few rules that can be useful for breaking cycles, that is, adding constraints involving totally new values does not effect the solution set of the constraints but also does not simplify finding a solution.
Algorithm 3: RANDOM SATISFACTION

Variables:
- TC [] is an array of values, indexed by the variables in the constraints.
- C is the system of constraints.
- N is a constant that limits the number of solutions to be attempted.
- c is a constraint.
- x is a variable.
- v is a value.

1. [Initialize]
   
   for each variable x in C
   
   \[ TC \{x\} = undefined \]
   
   end

2. repeat N times
3.   for each variable x in C
4.     find a random value v
5.     \[ TC \{x\} = v \]
6.   end
7.   Evaluate C on T [ ]
8.   if C is satisfied then
9.     return FEASIBLE
10. end
11. return no solution found

Figure 13. Random Satisfaction Procedure

If Algorithm 3 returns in line 9, then the test case in TC will satisfy the constraint. However, since each point in Algorithm 3 is chosen randomly, there is no guarantee that the test on line 8 will ever evaluate positively. So there is no guarantee that this procedure will find a solution. The likelihood that a randomly chosen test case chosen will satisfy C depends on the structure of the input domain and the density of satisfying test cases.

Table 10 shows the results of an experiment using random satisfaction on the program TRITYP. This experiment was run on an early version of the Godzilla implementation that used a random satisfier. As shown in the table, 276 test cases were created for TRITYP. The listing for TRITYP (TRiangle TYPE) is in Appendix B and the complete list of necessity and path constraints for TRITYP are shown in Appendix C. The random satisfaction procedure made 30,020 attempts to find the 276 successful cases. The last entry in the table is labeled Failures. This corresponds to the number of times the satisfier reached the maximum number of attempts \( N = 250 \) without successfully satisfying the constraints. In these cases, the satisfier gave up. These 107 failures corresponded exactly to the 107 equivalent mutants in TRITYP. These constraints were infeasible, which means that in this case the random satisfaction method always found a solution to the constraints where such a solution existed. In Chapter VII, the idea of using infeasible constraints to detect equivalent mutants is discussed in detail, with examples showing how the method works.

In the second column, the infeasible constraints are eliminated to give a more precise measure of how many random attempts are needed to successfully satisfy a constraint. Each of the infeasible constraints required 250 attempts before the satisfier failed, so there were 26,750 "useless" attempts. If we assume that we have some way to detect these infeasible constraints, then the satisfier would have
<table>
<thead>
<tr>
<th></th>
<th>All Mutants</th>
<th>Non-Equivalent Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Cases Created</td>
<td>276</td>
<td>276</td>
</tr>
<tr>
<td>Satisfaction Attempts</td>
<td>30,020</td>
<td>3,270</td>
</tr>
<tr>
<td>Attempts Per Test Case</td>
<td>109</td>
<td>12</td>
</tr>
<tr>
<td>Maximum Number of Attempts</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>Failures</td>
<td>107</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 10. Random Satisfaction Results for TRITYP

only taken 3,270 attempts, or an average of 12 attempts per test case. While we might wish to have a satisfaction method that only made one attempt per test case, the fact that the satisfaction process is automatic makes these results encouraging.

Although Godzilla employs no sophisticated infeasible constraint recognition techniques, we might hope for techniques that can sometimes detect infeasibility. For example, theorem proving techniques could possibly provide a partial solution. Without a way to detect the infeasible constraints in time to avoid useless test case attempts, these failures cause random satisfaction to be very expensive. The threshold value (maximum number of attempts) is very important to this technique. If set low enough, we can expect the satisfier to give up on the infeasible constraints very quickly. If set too low, then many feasible constraints will not be satisfied.

There is one argument for incorporating some randomness in the satisfier. Remember that the test case constraints are necessary, but not sufficient descriptions of a test case to kill a mutant. That is, even if we construct a test case that satisfies the constraints we are not guaranteed that it is the test case we want. So if we want to be sure of getting at least one effective test case, we may wish to generate several different test cases, each of which satisfies the constraints. Generating several satisfying test cases allows us to sample the space of test cases that satisfy the constraints. A procedure that generates the same (satisfying) test case every time would not have this quality. The next two procedures incorporate randomness just for this reason.

Propagate and Guess

The propagation procedure is described earlier in this chapter (Algorithm 2). Propagation is a greedy procedure that uses immediately available information to find values that satisfy the simplest constraints. It iteratively applies these partial solutions to complicated constraints until values have been found for each variable or until no more information can be deduced and the method fails.

When propagation fails it is generally because of cycles in the constraints. A pure propagation procedure would give up when cycles are found. The notion behind the propagate and guess procedure is to break these circular constraints arbitrarily—or guess at a solution. The reason this technique works when satisfying test case constraints is that we do not need a particular solution or an optimal solution, but any solution to the constraints. The cycle breaking step is simple. First some variable involved in the cycle is chosen and given a value arbitrarily. Then the other constraints become simpler and can be solved using the propagation method. If a cycle still remains, another variable is assigned and the process repeats.

Since the propagation procedure has already been listed (earlier in this chapter, Algorithm 2), it will not be repeated here. Algorithm 4 is meant to be included with propagation at the point where propagation fails, at line 25 in Algorithm 2. Rather than failing, the new value is chosen and the procedure restarts the loop at line 3.
Algorithm 4: PROPAGATE AND GUESS

Variables: (see Algorithm 2),
1. choose some constraint \( c \) in \( C \)
2. generate a random value \( v \) for some variable \( x \) in \( c \)
3. \( TC[x] = v \)
4. for every constraint \( c' \) in \( C \)
5.  if \( x \) is in \( c' \) then
6.     remove \( c' \) from \( C \)
7.     add \( c' \) to Next
8. end

Figure 14. Propagate and Guess procedure.

The arbitrary value assignment in line 2 can be viewed as adding an additional constraint to the system that requires that the chosen variable be equal to some constant. This new constraint could make a feasible set of constraints infeasible making the Algorithm fail (at line 13 rather than 25). As in the random satisfaction procedure, the entire procedure can be repeated until a satisfying set of values is found.

Assigning a random value to an arbitrary variable introduces randomness into the satisfaction procedure. This means that the procedure might not always find a solution when one exists. Heuristics such as choosing the constraint that has the fewest variables in common with other constraints in the system (the least chance for error) or the constraint that has the most variables in common with other constraints in the system (the most perturbation) can have some effect on whether a solution is found.

Propagate and Bisect

One way to solve the termination problem in the propagate and guess procedure is to apply a more intelligent search technique to find a successful assignment of values to the cycle. Bisection is a well-known search strategy often applied to numerical analysis problems (see e.g., chapter 2 of Numerical Analysis [Burd78]). Bisection views the input domain of the constraint variables as forming a geometric region and successively subdivides the region until either a successful variable assignment can be made or some stopping condition can be satisfied.

The geometric region is bounded by the constraints in the system that are not part of the cycle. Bisection begins by dividing the region into two approximately equal sub-regions. Then a point is chosen from each sub-region and the constraints are checked to see if either of the points satisfies the constraints. If the constraints are satisfied then the procedure stops. If not, then each of the sub-regions is bisected to form a new pair of regions. This recursive sub-division fails when the regions are point sized—indicating that there is no solution.

The bisection procedure is presented as Algorithm 5. It can also be incorporated with the propagation procedure at line 25. Rather than failing, propagation calls the bisection procedure with the remaining constraints.

The bisection procedure assumes that each variable has an associated "interval" that gives the current range of values that the variable can assume. This interval is divided into two equal sub-intervals at each step in the recursion. Bisection is presented as a recursive function that searches in a depth-first manner. A more efficient implementation would be as a breadth-first search; it is presented here as a depth-first search to simplify the presentation.
Algorithm 5: BISECTION

Bisect \((D, X, TC)\)

Variables:
- \(X = \) an array containing \(n\) unsatisfied variables.
- \(TC = \) an array of values indexed by the variables in the constraints.
- \(D = \) an array of domains, one for each variable. \(D(x)\) contains the (current) upper and lower bounds on \(x\), \(D(x) = (a_x, b_x)\).

1. [Initialize]
   for each interval in \(D\)
   \[ TC(x) = \frac{a_x + b_x}{2} \]
   end
2. evaluate \(C\) on \(TC\)
3. if \(C\) is satisfied then
   return FEASIBLE
4. else [do recursion]
5. for each variable \(x\)
6. \(D(x) = (a_x, TC(x))\)
7. if Bisect \((D, X, TC)\) returns FEASIBLE then
8. return FEASIBLE
9. \(D(x) = (TC(x), b_x)\)
10. if Bisect \((D, X, TC)\) returns FEASIBLE then
11. return FEASIBLE
12. end
13. end

Figure 15. Bisection Procedure

Bisection generally works efficiently if the constraints do have solutions. Since constraint systems describe geometric regions, solutions to the constraints tend to be adjacent to each other. That is, if a particular point does not satisfy the constraints, then its neighbors are not likely to satisfy them either. Bisection maximizes the distance between the points that are to be checked.

On the other hand, if there are no solutions to the constraints then bisection can be expensive. It will continue subdividing the region until all points have been tested. This is clearly impractical. The most obvious solution is to allow the recursion to go to some fixed depth. This succeeds if there is a satisfying region for the constraints that is too large to fit "between" the points that are chosen for evaluation on \(C\).

Bisection can be applied with or without randomness, depending on how a point is chosen within each sub-region. In the pure form (as presented above), the central point is used. This ensures maximal separation of the points that are tested for satisfaction. This also ensures that the same test case will be generated if the same constraints are satisfied more than once. Choosing the central point is not necessary to the technique and any point could be used. If a random point within each sub-region is chosen then different test cases can be generated by running the procedure multiple times.

Domain Reduction

The procedures above have all been studied and used many times. They are general purpose techniques that can be applied to a wide variety of problems. The approach taken in this section is to use
the techniques already described as a basis from which to design a procedure specifically designed for the types of constraints created for test data generation.

To start this discussion we first need to look at the particular characteristics of the problem. Constraint satisfaction is in general a very difficult problem [Baza77]. Since the constraints can be of arbitrary complexity, the problem is clearly as difficult as non-linear programming. On the other hand, test case constraints are seldom actually non-linear in form. A case study analysis of the 3600 test case constraints generated for five Fortran programs has shown that the constraints are usually simpler than what might be expected. The four characteristics below summarize this analysis. With each is given supporting data from the constraint analysis.

1. The test case constraints are almost always linear. Of the 3600 constraints studied, none were non-linear.

2. The test case constraints usually have a simple form. That is, there are few conjuncts and disjuncts in the expressions and they involve only a small number of variables. In the constraints studied, there were less than two disjuncts per constraint system. Recall that the constraints in Godzila are kept in disjunctive normal form. The disjuncts had about five conjuncts apiece and the conjuncts averaged about 1.5 variables.

3. Many constraints involve inequalities rather than equalities. This is a subtle but important advantage. Though we think of constraints as describing continuous, perhaps infinite regions, a computer program must necessarily find solutions over discrete, finite regions. Because of this discrete nature, equality constraints are satisfied by a smaller set of points than inequality constraints. So inequalities constraints are easier to satisfy. The constraints studied also support this observation. Only about 10% of the constraints are equality constraints.

4. The constraints contain many constants. Constant values simplify the constraints and are easier to handle algebraically. Constants are not involved in cyclic dependencies and provide starting values for the propagation procedure. In the constraints analyzed, there were about two constants for every three constraints and seven constants per disjunctive clause.

The fact that test case constraints have a simple form is not magical or accidental. The path expression constraints generated from a program are constructed by data flow analysis. Allen and Cocke [Alle76] describe a data flow procedure that works by reducing the data flow graph of a program, a process similar to constraint satisfaction. Even though they state that some data flow graphs cannot be reduced by their method, they also point out that these irreducible graphs were very rare. In 75 Fortran programs that they studied, they found that over 90% of the data flow graphs were reducible. In a similar study, Knuth [Knut71] looked at 50 Fortran programs, all of which were reducible. It seems likely that programs that are considered "structured" are usually reducible. The examples of irreducible program flow graphs that Allen and Cocke [Alle76] contrived involved such things as "crossed loops" and "backwards goes", both of which are considered poor programming structures [Kern78]. This type of programming also results in complicated path expressions. Since most of the complexity of the test case constraints comes from the path expressions rather than the necessity conditions, these studies provide some support for the fact that the constraints are usually simple.

The characteristics described suggest that a simple and fast procedures might be effective at satisfying the types of constraints found in a test case generator. The approach taken was a modification to the propagation procedure. The main principle behind this technique is to delay variable assignment

---

2. These are the programs that are analyzed in Chapter IX. They are listed in Appendix II.
until as much information is known as possible. Then when we do have to make guesses about a value for a variable, we have already eliminated many incorrect values.

It is easiest to describe domain reduction in a geometric description manner. The test case constraints define some N-dimensional space of input values (the satisfying space). Each time a variable is assigned a value, this space is reduced by one dimension. As the number of dimensions reduces, the constraints in the system become progressively simpler. Each assignment also introduces a new constraint into the system of the form \( x = k \), where \( k \) is some constant. If chosen poorly, this new constraint may make the region infeasible. The assumption is that because of the simple form of the test case constraints, these dimension reducing constraints will seldom make the region infeasible. When they do, the infeasibility will be easily recognized. Infeasible regions after the dimensions of the constraints have been reduced will show up as conflicting constraints such as:

\[
(X = 4, \ X > 8)
\]

The domain reduction procedure follows. It uses an array of domains, much like the intervals used in the bisection procedure. The procedure uses the constraints to successively reduce the size of each variable's domain until a value is assigned. For each variable, a constant \( \varepsilon \) is assumed to exist, where \( \varepsilon \) is the smallest separation between two values of the variable.

**Algorithm 6: DOMAIN REDUCTION**

**Variables:**
- \( TC \) is an array of values, indexed by the variables in the constraints.
- \( C \) is the system of constraints.
- \( x, y \) are variables.
- \( D \) is an array of domains, one for each variable. \( D[x] \) contains the list of (current) upper and lower bounds on \( x \), \( D[x] = (a_x: b_x) \).
- \( R \) is a list of newly resolved variables.
- \( e \) is a constraint.
- \( v \) is a value.
- \( k \) is a constant.
- \( \gamma \) is a conditional operator (\( >, <, =, \geq, \leq, \neq \)).

1. [Initialize]
   \( X = 0 \)
   
   for each variable \( x \) in \( C \)
   
   \( TC[x] = \text{undefined} \)
   
   end
   
   \( R \) is empty

2. repeat
   3. for each \( x \) in \( R \)
   4. for each \( e \) in \( C \)
   5. replace \( x \) in \( e \) with the value \( TC[x] \)
   6. end
   7. end
   8. for each \( e \) in \( C \) of the form \( x \gamma k \)
   9. adjust \( D[x] \) according to the operator:
   10. if \( \gamma \) is "\( > \)"
   11. \( D[x] = (k : b_x) \)
   12. if \( \gamma \) is "\( < \)"
   13. \( D[x] = (a_x : k) \)
   14. if \( \gamma \) is "\( \neq \)"


\[ D \{ x \} = (a_x, k - \epsilon, k + \epsilon, b_x) \]

16. if \( \gamma \) is "<" then 
17. \[ D \{ x \} = (a_x, k - \epsilon) \]
18. if \( \gamma \) is ">" then 
19. \[ D \{ x \} = (k - \epsilon, b_x) \]
20. if \( \gamma \) is "=" then 
21. \[ D \{ x \} = (k, k) \]
22. if \( a_x > b_x \) then 
23. return INFEASIBLE 
24. if \( a_x = b_x \) then 
25. \( TC \{ x \} = k \)
26. place \( x \) in \( R \)
27. remove \( \epsilon \) from \( C \)
28. end 
29. until \( R \) is empty 
30. for each \( c \) in \( C \) of the form \( x \gamma y \) 
31. fix the value of \( x \) to be \( v \), where \( v \) is in the domain \( D \{ x \} \), \( (a_x \leq v \leq b_x) \)
32. \( TC \{ x \} = v \)
33. \( \epsilon = v \gamma y \)
34. adjust \( D \{ y \} \) as in lines 10 through 21 
35. if \( a_y > b_y \) then 
36. return INFEASIBLE 
37. if \( a_y = b_y \) then 
38. \( TC \{ y \} = k \)
39. place \( y \) in \( R \)
40. remove \( \epsilon \) from \( C \)
41. end 
42. if \( R \) is not empty then 
43. goto line 2 
44. if there is some variable \( x \) such that \( TC \{ x \} = \text{undefined} \) then 
45. fix the value of \( x \) to \( v \), such that \( v \) is in the domain \( D \{ x \} \)
46. \( TC \{ x \} = v \)
47. place \( x \) in \( R \)
48. goto line 2 
49. else 
50. return FEASIBLE 

Figure 16. Domain Reduction Procedure

If the procedure terminates at line 50, then the test case in \( TC \) satisfies the constraints. If the procedure terminates at either line 23 or 36, then the current test case (in \( TC \)) does not satisfy the constraints. This implies that the original constraint system is infeasible only if neither line 31 nor line 45 was executed. Since these lines make arbitrary assignments of values to variables, then they may introduce infeasibility to an originally feasible constraint system. If there are \( M \) variables and \( N \) constraints, the loop in line 1 is executed \( M \) times. The loop from line 8 to line 28 executes at most \( N \) times. Since each time the loop from line 2 to line 29 is executed, some variable \( x \) is replaced by \( TC \{ x \} \)
in each constraint, the loop can be executed at most $M$ times. Thus, the upper part of this procedure runs in time proportional to $M \cdot N$. The loop from line 30 to 41 executes $N$ times. If the procedure returns to line 2 from line 43 or 48, it does not execute the repeat loop another $M$ times. Because a variable can only be placed into $R$ one time throughout execution, this does not increase the running time.

Algorithm 6 works well on constraints when a majority of the constraints are of the form $x \gamma k$ or $x \gamma y$. In the test case constraints studied, 58% were of the form $x \gamma k$, and 30% were of the form $x \gamma y$. This heuristic of evaluating the simpler constraints first lets known values propagate through the constraints before variables need to be assigned arbitrarily. Moreover, by keeping track of the domains in $D$, the procedure essentially encodes (and combines) constraints in a way that eliminates values that do not satisfy the constraints without eliminating values that do. When arbitrary decisions about a variable’s value are made (in lines 31 and 45), many values that do not satisfy the constraints have already been eliminated—leaving a higher density of values that do satisfy and less room for error.

As an example of satisfying constraints with the domain reduction procedure, consider the following constraints:

$$(a = 15) \land (b \geq a) \land (b \leq c) \land (d \leq c + 15) \land (e \leq f + g)$$

For this example, all domains are assumed to be initialized to $(-100:100)$. The first step would be to use the value for $a$:

$${\text{TC}}\ [a] = 15$$

This allows the system to be reduced:

$$(b \geq 15) \land (b \leq c) \land (d \leq c + 15) \land (e \leq f + g)$$

The constraint $b \geq 15$ is then merged into the domain for $b$:

$$D\ [b] = (15:100)$$

This leaves:

$$(b \leq c) \land (d \leq c + 15) \land (e \leq f + g)$$

At this point, a value for either $b$ or $c$ must be fixed. If the heuristic to fix the variable with the smallest current domain is used, then an arbitrary value for $b$ is taken from the range 15 to 100:

$${\text{TC}}\ [b] = 34$$

Then the constraints become:

$$(34 \leq c) \land (d \leq c + 15) \land (e \leq f + g)$$

The constraint $34 \leq c$ is then merged into the domain for $c$:

$$D\ [c] = (-100:34)$$

So the remaining constraints are:

$$(d \leq c + 15) \land (e \leq f + g)$$
Now a value must be fixed for one of the remaining values. Again, the heuristic of fixing the variable with the smallest current domain is used. So an arbitrary value for \( c \) is taken from the range -100 to 34:

\[
TC [c] = -10
\]

This leaves:

\[
(d \leq 5) \land (e \leq f + g)
\]

The domain for \( d \) is updated:

\[
D [d] = (-100; 5)
\]

So the remaining constraint is:

\[
(e \leq f + g)
\]

Since there have been no other constraints involving either \( e \), \( f \), or \( g \), this constraint has not been updated. Choosing an arbitrary value for \( f \) gives:

\[
TC [f] = 29
\]

\[
(e \leq 29 + g)
\]

Again, an arbitrary value is chosen, this time for \( g \):

\[
TC [g] = 36
\]

The remaining constraint is:

\[
(e \leq 65)
\]

This is used to modify the domain for \( e \):

\[
D [e] = (65; 100)
\]

Now that all the constraints have been eliminated, we are left with values for the variables \( a \), \( b \), \( c \), \( f \), and \( g \) (15, 34, -10, 29, 36). The variables \( e \) and \( f \) are still undefined—but since there are no more constraints, they can take any value within their current domain.

The choice of an initial domain for the variables can increase the efficiency of the procedure by reducing the size of the search space. For example, if it is known that the program being tested does not need to accept negative values, the initial domain may as well start at zero. If the first step in the test program is to test for negative inputs then we may wish to use some negative numbers as inputs, but the majority of the test cases will require positive numbers. The technique will converge slightly more quickly if the initial domain is adjusted to include only a small number of negative values. Since the domain does need to be at least large enough to contain all the constraints, limiting the domain is difficult to do automatically and should probably be done manually by someone who is familiar with the test program.