Experimental Results from an Automatic Test Case Generator

RICHARD A. DEMILLO
Purdue University
and
A. JEFFERSON OFFUTT
Clemson University

Constraint-based testing is a novel way of generating test data to detect specific types of common programming faults. The conditions under which faults will be detected are encoded as mathematical systems of constraints in terms of program symbols. A set of tools, collectively called Godzilla, has been implemented that automatically generates constraint systems and solves them to create test cases for use by the Mothra testing system. Experimental results from using Godzilla show that the technique can produce test data that is very close in terms of mutation adequacy to test data that is produced manually, and at substantially reduced cost. Additionally, these experiments have suggested a new procedure for unit testing, where test cases are viewed as throw-away items rather than scarce resources.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*test data generation*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Adequacy, constraints, mutation analysis

1. INTRODUCTION

This paper describes experimental results that are based on a new technique for generating test data. This technique, called constraint-based testing (CBT), uses the source code to automatically generate test data that attempts to satisfy the mutation adequacy criteria. Elsewhere, we describe the technique [9, 10] and the details and algorithms of the implementation [29]; here we describe a set of experiments that measure CBT.

Some of this work was done while the authors were at the Georgia Institute of Technology. This research was supported in part by contract F30602-85-C-0255 through the Rome Air Development Center.

Authors' addresses: R. A. DeMillo, Software Engineering Research Center, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907; A. J. Offutt, George Mason University, ISSE Department, 4400 University Drive, Fairfax, VA 22030; email:ofut@gmuvax2.gmu.edu

From these experiments, we suggest a novel approach to using mutation for unit testing, where test cases are viewed as "throw-away" items rather than expensive, scarce resources. With this view, test cases are automatically generated and immediately executed against the mutants. This is repeated until the test cases kill no more mutants. At this point, all test cases that killed no mutants (with current technology the majority of the test cases generated) can be eliminated. Up to this point, the process is entirely automatic. To finish testing, the tester will manually examine the output of the test cases to determine whether the outputs are correct and either add test cases to kill the remaining mutants or decide that they are equivalent. By postponing these manual steps, generating test data automatically adds very little expense to the mutation process and removes a great deal of expense.

Program mutation guides the testing process by measuring test data quality. A test set is *adequate relative* to a set of faults if it distinguishes the test program from a set of incorrect programs represented by the faults. These fault-laden programs are called *mutants* of the program being tested [12]. It is sometimes useful to record the extent to which a program has been so distinguished. This is done by "marking" mutants either dead, alive, or equivalent and by computing a *mutation score*, which is the percentage of nonequivalent mutants that are dead. A test set is *mutation adequate* for a program if its score is 100%.

Unfortunately, generating mutation adequate tests can be labor intensive. To generate these tests, a tester interacts with an interactive mutation system (the most recent of which is the Mothra software-testing environment [10, 21]) to examine remaining live mutants and design tests that kill them. Our strategy for automatically generating test cases is to represent the conditions under which those mutants die as mathematical systems of constraints on the definition of tests and to automatically generate values that satisfy the constraint systems. We create our constraint systems in a *conservative* fashion, by creating constraints that may be satisfied by test cases that do not kill their mutants instead of constraints that exclude test cases that will. Our ultimate goal (which is, of course, theoretically unreachable) is to completely automate the testing and measurement process so that when a correct program is submitted for testing, a mutation-adequate test set is produced as evidence of correctness without human intervention; and when an incorrect program is submitted the tester is provided—again, without human intervention—a set of tests on which the program fails and a list of possible faults and their locations suitable for input to a debugging tool.

The work described here is part of the Mothra software-testing environment project [10]. An implementation of the CBT technique, Godzilla, has been integrated with Version 1.5 of the Mothra testing system. We have carried out several experiments to measure the quality of the test data generated by Godzilla, to measure the performance of the technique and the tool, and to discover ways in which CBT could be improved. The paper begins with an overview of mutation testing followed by a description of the CBT test data generation technique. We present the rationale for CBT in terms of mutation testing, since the two methods share a common underlying theoretical basis. Following this background material the experiments are described. They were conducted using the Godzilla test data generator to generate test data and the Mothra system to run the experiments. Finally, results, implications, and discussions for improvements are presented.

2. OVERVIEW OF MUTATION

Mutation testing is a fault-based testing method that measures the quality of a set of independently created test cases [10, 12]. In practice, a tester interacts with an automated *mutation system* to determine the adequacy of a test data set and to improve that test data. The mutation system forces the tester to test for specific types of faults, which are represented by simple syntactic changes to the test program that produces mutant programs.

For example, function MAX is shown in Figure 1 with four first-order mutants embedded in the program (lines preceded by a "Δ"). The convention when displaying mutants is to show the original program with each mutated statement inserted immediately below its corresponding original statement. Each mutated statement represents a separate (mutant) program with the original statement replaced by the mutated statement. MAX returns the larger of its two integer inputs.

Mutants are designed to represent common mistakes that programmers make.[1] The goal of the tester when using a mutation-testing system is to create test cases that cause each mutant to fail. First a test case is run on the original program, and the output is saved as the *expected output* of that test case. Then, the tester examines the output and decides if it is correct; this is called *expected-output examination*. If the output is incorrect, an error has been discovered; the program must be changed; and testing must start again. If the expected output is correct, the test case may be run against mutants. When the output of a mutant differs from the expected output of the test case, that mutant is considered *dead*, and the test case is said to have *killed* it. For example, the input ($M = 2$, $N = 1$) produces the expected output $MAX = 2$. On the first mutant in Figure 1, the output is $MAX = 1$. Since the mutant output is incorrect, this test case kills that mutant, demonstrating that the original program does not contain the fault that is equivalent to the mutant. Because the coupling effect [12, 30] tells us that test cases that cause simple faults to result in failure also cause complex faults to result in a failure, this also demonstrates that the original program does not contain the complicated faults represented by the mutant.

Each mutant is executed on each test case until it is killed or passes all tests. A set of test cases that kills all mutants is adequate relative to that set of mutants. A program that has been successfully tested with an adequate test set is either correct or contains a fault that has not been represented by

[1]Mutation testing is partially based on the empirical *competent-programmer assumption*, which states that though programmers create erroneous programs, the programs they create are syntactically close to being correct.

Fig. 1. Function MAX.

```
     FUNCTION MAX (M,N)
     MAX = M
Δ    MAX = N
Δ    MAX = ABS (N)
     IF (N .GT. M) MAX = N
Δ    IF (N .LT. M) MAX = N
Δ    IF (N .GE. M) MAX = N
     END
```

the mutants. The theoretical properties of adequacy have been discussed by Budd and Angluin [3], Gourlay [18], Morell [26], and Weyuker [37].

It is generally impossible to kill all mutants of a program because some changes have no functional effect. The last mutant shown in MAX is an *equivalent* mutant that can never be killed because it always produces the same output as the original program. In current mutation systems, equivalent mutants are recognized by human examination or by relatively primitive heuristics [1, 8]. In fact, a complete solution to the equivalence problem is not possible [3]. Recognizing equivalent mutants and creating 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. Although the focus of this paper is on using CBT to generate test data, Offutt's dissertation [28] suggests using infeasible constraints as a partial solution to the problem of detecting equivalent mutants.

3. AUTOMATICALLY GENERATING TEST DATA

Practical test data generation techniques choose a subset of all possible inputs according to some testing rationale [34]. Constraint-based testing tries to choose a mutation-adequate set of test cases by describing the conditions necessary to kill each mutant as a system of mathematical constraints. Although this problem is, in general, unsolvable, we apply partial solutions to generate test data sets that approximate mutation adequacy. Inevitably, these partial solutions cause us to create constraint systems that are either too strong (excluding test cases that will kill the mutant) or too weak (including test cases that will not kill the mutant). We take a conservative approach to constraint creation and allow constraints that may be satisfied by test cases that will not kill the mutant. Thus, each constraint system describes a set of test cases that includes all test cases that will kill the mutant, as well as some that will not. Each constraint system is solved to generate a test case that *targets* the mutant. BSEARCH is a small example shown in Figure 2 with one embedded mutant on statement 13, a *variable replacement* mutant that replaces *MID* by *HIGH*.

To understand how to select a test case to target a mutant, we must first recall that the mutant is represented as a syntactic change to one statement. Since every other statement in the mutated program is the same as in the original program, it is apparent that as a minimum we must execute

```
            LOGICAL FUNCTION BSEARCH (LIST, ELEM)

            INTEGER LIST (10), ELEM, LOW, HIGH, MID
1           LOW = 1
2           HIGH = 10
3       10  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
Δ                 LOW = HIGH + 1
14              ELSE
15                HIGH = MID - 1
16              ENDIF
17              GOTO 10
18            ENDIF
19          ENDIF
            RETURN
```

Fig. 2. Mutant of BSEARCH.

the mutated statement. In addition, the test case must cause the mutant to have an incorrect state after some execution of the mutated 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 an output that reveals the fault. We discuss each of these points in more detail below.

3.1 Necessity

The *necessity condition* is the key idea behind constraint-based testing. The *state* of a program consists of the values of program variables and internal variables such as the program counter. To kill a mutant, a test case must create some state in the mutant program that is incorrect with respect to the original program. Since the mutant and original programs are identical except for the mutated statement, if the states of the two versions of the program are equal after the last execution of the mutated statement, then the mutant will not die. Thus, the *necessity condition* requires that the state of the mutated program be incorrect after **some** execution of the mutated statement. In line with our conservative approach, we do not require the state to be incorrect after the first, or the last execution, but after **any** execution.

The necessity condition formalizes the requirement of Howden's weak mutation [20]. In weak mutation, the states of the original and mutant programs are compared over a "component" of the program that contains the mutated statement. If we view the portion of the program executed up to and including some execution of the mutated statement as a component, then the necessity condition forces the mutated component to produce different output from the original component. This is exactly the requirement of weak mutation. Of course, the original idea of weak mutation, as implemented by Girgis and Woodward [15], was to avoid executing mutants by analytically determining under what conditions the mutants would die. Although this kind of analytical approach is computationally less expensive than an execution-based approach, and was what Howden originally intended, it suffers from two problems. First, whether a mutant can be killed can only be obtained for a few mutant types. For example, Girgis and Woodward [15] were only able to handle three mutant types (wrong variable, off by a constant, and wrong relational operator), of the 22 that Mothra uses. These additional operators allow for considerably more fault detection power than systems that use a small subset of them. Second, since no separate executions are being done for the mutants, the components must have a very localized extent, precluding the ability to determine the status of the mutant under strong mutation.

Necessity constraints have been derived from the Mothra mutant types [9, 28]. They are generated by using the same algorithm used by Mothra's mutant maker [21] except that instead of mutants, constraints are produced. As each program symbol or expression is encountered, each mutation operator that can be applied to that symbol is used to determine the conditions under which that mutant will cause an incorrect intermediate state. These conditions are modeled as algebraic constraints on the test cases.

For example, in BSEARCH, the value of LOW after statement 13 in the mutant 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, the necessity constraint for this mutant requires that MID and $HIGH$ have different values ($MID \neq HIGH$). The algorithm for generating necessity constraints is given elsewhere [29].

3.2 Reachability

Although generating test cases that are guaranteed to reach a particular statement is generally undecidable, we approximate this goal by creating a *path expression* constraint (PE) for each statement that describes all paths, up to but not including loops, to that statement. Each sequential path is represented by a separate disjunctive clause; since we are only concerned with *reaching* a statement, intervening loops are represented by describing the conditions necessary to be executed through the loop some number of times. Path expressions are well known (e.g., [5, 13]) and describe a condition or conditions on a test case that are necessary for a program to reach a statement. Creating constraints that are only necessary for reachability is in line with our conservative approach, since this does not exclude any test cases that will reach the statement.

Following is a summary of Godzilla's path expression algorithm, which is given in [29]. The current path expression (*CPE*) constraint system describes the conditions necessary for a test case to reach the program statement currently being processed. The current statement is S, and PE_S is the path expression to S. When the algorithm terminates, PE_S contains a sequence of disjunctively connected constraints, where each disjunct represents a distinct path to S. The *CPE* is initially given the default value TRUE, and the path expression for each statement is given the value FALSE, indicating that no path to that statement has been found. Godzilla examines each statement in turn, performing several actions for each statement. First, the *CPE* is ored to the previous path expression for S, representing a new way to reach S, and each path to S is represented as a new disjunctive clause in the path expression. Next, the *CPE* gets the new PE_S, so that each disjunctive clause in the *CPE* represents a different execution path through the program to the statement following S. Finally, if S is a branch statement, the *CPE* is updated by a modification rule that reflects the branch in S (the detailed set of rules are given in [29]), and the branch predicate of S is conjuncted with the *CPE* and added to the path expression of the target statement of S's branch.[2]

To kill the mutant in 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 requires that *HIGH* be greater than or equal to *LOW*, *ELEM* not be equal to *LIST*(*MID*), and *ELEM* be greater than *LIST*(*MID*). These constraints are later rewritten to be in terms of input variables, where possible, using symbolic evaluation (described in Section 3.4).

When completed, a path expression for a statement represents several ways of getting to that statement. Godzilla represents each path to a statement as a disjunctive clause in the path expression constraint. In fact, the disjunctive clauses represent all paths to a statement up to, but not including, loops. Since loops generate an unknown or unlimited number of paths to a statement, Godzilla forms a constraint clause that will cause at least one iteration through the loop. For example, if the loop body is DO 10 I = M, N, Godzilla will form the constraint $N \geq M$. Since the goal of reachability is merely to ensure that a statement is reached, a test case must execute the loop body once, but it does not matter how many additional times the loop is executed. Likewise, the case of executing the loop body zero times is irrelevant to Godzilla, because to reach statements after the loop, it does not matter how many times the loop is executed. Both of these approaches are in line with our conservative approach, since they do not exclude any test cases that will reach the statement. Which path will be executed depends on which disjunctive clause is satisfied to generate test cases. This depends on the satisfaction procedure and involves factors such as the order of the

[2]The version of Godzilla used for these experiments incorporated no backtracking, which limited its ability to handle programs with backward GOTOs. A recent modification [32] has used data flow techniques to handle the backtracking problem.

disjunctive clauses, the necessity constraint, the complexity of the constraint system, and the randomness inherent in the satisfaction procedure.

3.3 Sufficiency

Although an incorrect local state is necessary to kill a mutant, it will not always guarantee that the final state of the mutant will be incorrect. Once the states of the mutant and the original program diverge, they may well converge to the same, final state. So the constraint system 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.

It will have occurred to the reader that our constraint systems are weak in a certain formal sense. Whereas it might be possible to generate constraints the solution of which would be a sufficient condition for killing mutants, it is entirely possible to satisfy our constraints without ever generating a test set that is useful for mutant elimination. This limiting feature of our tool represents a design compromise.

A sufficiency constraint implies reachability—that is, if the constraint is satisfied then the program reaches the end state with its internal variables having some specific properties. If there were inductive assertions at each loop in the program, it would be possible to synthesize the sufficiency constraints. Without inductive assertions, synthesizing the constraints necessarily leads outside the language that we are using to build constraints and would in, any event, result in exponentially large and therefore unmanageable constraint systems (see, e.g., Manna [23]).

Furthermore, sufficiency constraints are really partial-correctness conditions [23]. "Solving" such constraints is a highly intractable problem—definitely beyond the solution of the simple constraint systems that make up our language. General systems of constraints are not likely to be algorithmically solvable. Good heuristics for nontrivial special cases are not even available [33].

In light of such difficulties we have decided to first explore the effectiveness of using the reachability and necessity constraints alone. By applying this simplifying strategy, we have found that test cases that solve the reachability and the necessity conditions also satisfy the sufficiency condition frequently enough to be useful in practice.

The experiment in Section 4.2 is designed to estimate the probability of solving a sufficiency constraint given a solution to reachability and necessity constraints.

Independent support for this approach has also been given by two recent experimental studies of weak mutation [24, 31], both of which found that test sets that cause incorrect local states for all mutants are strong enough to cause over 90% of the mutants to have incorrect final states.

The necessity condition is equivalent to Morell's *creation* condition, and the sufficiency condition is equivalent to Morell's *propagation* condition [25, 26] and Richardson and Thompson's extensions [36]. The distinction is that whereas Morell's creation condition describes program states in which a set of mutants would alter the state of the program [25], the necessity condition describes a test case that will cause an incorrect state in **one** mutant of the program. Although less general, our necessity conditions are directly modeled as mathematical constraints, and we automate the discovery of the reachability condition.

3.4 Constraint Reduction

An *input* variable has a value included for it as part of a test case. A variable is *internal* to the program if it is not part of the input set. Internal variables are undefined, when a program begins execution, and are assigned values during execution (or not at all). Although values for internal variables are not included in a test case, they may still appear in constraint systems. The *internal-variable problem* is that of generating a test case that will cause some internal variable to take on a specific value at a particular point in the program's execution. Since internal variables are based on program inputs, they can be controlled indirectly. This problem is not new to constraint-based testing techniques and is encountered in path-testing strategies, among others. Internal variables can be approximately described symbolically in terms of input variables using symbolic execution techniques of the sort previously used in data flow testing [7, 22, 35]. Such techniques are not able to completely determine values within loops and for arrays, but are often able to make useful approximations. Godzilla deals with loops by weakening constraint systems, which yields test cases that are less powerful, but still useful. Godzilla deals with arrays by treating the first two array elements as separate variables; again, this results in weaker, but still useful, constraint systems.

Godzilla computes, for each variable, its symbolic value on each statement in the program [29]. These symbolic values are lists of ranges of constant values (initially infinite) that the variable may take on, along with the constraint systems that describe under what conditions the variable will have a value from the corresponding range. The conditions are developed from the path expression disjuncts for the statement. For a given statement S, a variable X will have at most one value for each execution path that reaches S. Thus, each symbolic value for X has an associated path expression. Symbolic evaluation is used to determine the symbolic representation of each of the values that X may have at S. These are used to reduce X to be in terms of input variables only.

3.5 Solving Constraints

Finding values to satisfy constraint systems is a difficult problem that arises in such diverse areas as computability theory, operations research, and artificial intelligence. Godzilla employs heuristics that work efficiently and produce satisfying test cases when the constraint systems have a simple form. These heuristics are intuitively described here; the full algorithm, with justification for why they work for test case constraint systems, can be found elsewhere [29].

The "domain reduction" satisfaction procedure that Godzilla uses is a modification of the propagation algorithm [17], which is in turn based on the well-known topological sort algorithm. The domain reduction procedure uses local information in the constraint systems to find values for variables, then uses back substitution to simplify the remaining constraints in the constraint system. Each simplication eliminates one variable in the constraint system and also reduces the domain of values that other variables can take on. When this process halts (when no more simplification can be done), a heuristic is employed to choose a variable to assign a value to. This value is chosen randomly from the current domain of values that the variable can take on. This value is then back-substituted into the remaining constraints, and the process is repeated until all variables have been assigned a value.

This procedure is called the domain reduction procedure because the domain of values that each variable can take on is repeatedly reduced until an assignment is made. Domain reduction succeeds when constraint systems are simple and have linear equations and terminates even with more complicated constraints, although it will not always generate a solution if the constraint system is infeasible or if there are relatively few test cases that will satisfy the constraint system. Domain reduction employs randomness as part of the heuristics. This randomness is important because we sometimes wish to satisfy constraint systems several times to generate different test cases for the same mutant.

4. EXPERIMENTS WITH GODZILLA

We present five studies that measure the test data generator in terms of the effectiveness of CBT, the quality of the solutions to the technical problems, and the quality of Godzilla. A suite of seven Fortran-77 programs was used for these studies. The seven programs were taken from the literature and chosen to represent different types of problems to exercise the generation capabilities in as wide a manner as possible. The seven programs are BSEARCH, BUBBLE, DAYS, EUCLID, FIND, PAT, and TRITYP.

BSEARCH is shown in Figure 2. Since it requires its input array to be sorted, we added a call to a sort procedure as the first statement (but only tested the BSEARCH procedure). BUBBLE implements the bubble-sort algorithm on a numeric array. DAYS is taken from Geller [14] and Budd's dissertation [2] and computes the number of days between two given dates. EUCLID computes the greatest common denominator of two integers. FIND was studied by Hoare [19] and by DeMillo et al. [12]. It accepts an array A of integers and an index F. It returns the array with every element to the left of $A(F)$ less than or equal to $A(F)$ and every element to the right of $A(F)$ greater than or equal to $A(F)$. PAT is a pattern matcher that decides if a pattern can be found in a subject. TRITYP has been widely studied in software testing [6, 12, 34] etc., and takes three integers as input that represent the relative lengths of the sides of a triangle and classifies the triangle as equilateral, isosceles, scalene, or illegal. The versions we used have been recently rewritten and/or translated from earlier versions, removing previously reported bugs (e.g., in DAYS and TRITYP).

These experiments were conducted using the Mothra mutation-testing system version 1.5 running on Sun workstations running SunOS version 4.1.

4.1 Test Case Adequacy

Since the rationale behind constraint-based testing is mutation adequacy, the most direct measurement of CBT is to calculate the mutation score of the test data. If (1) the total number of mutants is M, (2) the number of dead mutants is K, and (3) the number of equivalent mutants is E, then the mutation score is

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

The mutation score is a quantitative measure of not only how well the test data approximates adequacy, but also, assuming mutation testing is an effective technique, of how well the data tests the program [4, 10]. For this experiment, test cases were generated by Godzilla for each of the programs in the suite; all mutants were generated; and each test case was executed against all live mutants. This is the typical way that testers use Mothra. The results of this experiment are displayed in Table I (preliminary results from this experiment were reported earlier [9]). The first column, *TC*, is the number of test cases generated; the columns M, K, E, L, and MS are the total number of mutants, the number of killed, equivalent, and live mutants, and the mutation score. The equivalent mutants for each program were determined by hand. The last column, *Time*, represents the wall clock time in minutes and seconds that Godzilla took to generate the test cases (on an unloaded Sun workstation).

Godzilla generated test data with a mutation score of over 0.97 for each program. Practical experience has shown that it is difficult and time consuming to manually create test data that scores above 95 on a mutation system. For example, students in software-testing classes typically spend 10 to 15 hours constructing test cases to kill the mutants for TRITYP. Given the automatically produced test data, it only required a short time to manually find test cases to kill TRITYP's remaining 19 mutants and to determine equivalent mutants.

Godzilla also generates many more test cases than a human tester does. Expected-output examination of this large number of test cases can be very time consuming. On the other hand, most test cases that are executed early in the mutation process kill a large number of mutants, including mutants targeted by test cases that have not yet been executed. As a result, only a few of the automatically generated test cases actually kill a mutant. An *effective* test case is one that kills at least one mutant (not necessarily its target); in the adequacy experiment, less than 10% of the test cases were effective. Thus, although Godzilla is very effective at killing mutants, it is grossly inefficient.

Table I. Test Case Adequacy Results

Program	*Test Cases*	*Mutants*	*Killed*	*Live*	*Equivalent*	*Mutation Score*	*Time*
BSEARCH	235	289	260	2	27	0.99	0:21
BUBBLE	314	338	303	0	35	1.00	0:26
DAYS	2637	3010	2783	88	139	0.97	6:47
EUCLID	163	195	171	0	24	1.00	0:09
FIND	455	1022	934	0	88	1.00	1 31
PAT	359	513	445	0	68	1.00	1 19
TRITYP	713	951	822	19	110	0 98	2 37

We are currently exploring ways to improve Godzilla's efficiency without reducing the strength of its test case sets. Our current method is to reduce the number of test cases after mutant execution. Since the goal of Godzilla is to generate test cases that satisfy the mutation criteria, we can solve this problem by ignoring any test cases that do not kill mutants. If the tester examines the output of only the effective test cases, the human intervention time is not significantly more than if the test cases were generated manually. Mothra can delete all ineffective test cases and allow the tester to examine expected output after mutant execution. This approach limits the amount of time doing manual tasks by increasing the machine time.

4.2 Test Case Precision Experiment

The adequacy experiment measures the test data set as a whole, following the typical way testers used mutation. On the other hand, each test case is constructed with a specific goal—to kill its target mutant. The *overkill effect* refers to test cases that not only kill their target mutants but also kill mutants they were not targeted for. A logical question is: how many test cases are able to kill their target mutants.

To measure the test cases individually, we executed each test case against only its target mutant(s).[3] We call this a "test case precision experiment" because it measures how precise each test case is at killing its target mutants. Note that with this strategy, a test case must kill its mutant to be effective. A test case may miss its target mutant because the sufficiency condition was not satisfied (and needed to be), because the path expression constraint was not strong enough (due to undecidability problems), or because the necessity constraint was not strong enough (this has become less common as necessity constraints have been enhanced). If M' mutants are targeted by some test case and if K is the total number of target mutants killed, then the precision is

$$P = \frac{K}{M'}.$$

[3]When two mutants result in identical constraint systems, only one test case is generated, and this test case is targeted for both mutants. Ideas for expanding this notion to satisfy nonequivalent constraint systems with the same test case are described under Combination of Test Case Constraints (Section 4 5).

Table II. Precision Experiment Results

	Test Cases	*Precision*
BSEARCH	235	.40
BUBBLE	314	.87
DAYS	2637	.35
EUCLID	163	.89
FIND	455	.71
PAT	359	.20
TRITYP	713	.57

Table II shows the precision of the test cases that were used for the data in Table I. As explained in the previous paragraph, there are several reasons why a test case may not kill its target mutant, so P is lower than if test cases were ineffective only because of sufficiency. This also means that this measurement cannot be viewed as a useful measure of the weak mutation hypothesis. Since the mutation scores for the five programs were significantly higher than the precisions, this table shows us that the overkill effect is quite important. Unfortunately, it also shows a wide variation in the precision over even a small number of programs.

Although each additional test case incurs additional cost during testing, we can anticipate that only a few test cases will be needed per constraint system. Each time we satisfy a constraint system, the test case has some probability (say ρ) of killing the mutant. Since our domain reduction procedure selects random data points within the space described by the constraint system [29], satisfying a constraint system several times allows us to make independent choices from the set of test cases that satisfy the constraint system. We have investigated the randomness of the procedure through χ-square tests with positive results.

Note that ρ is a random variable whose distribution depends on the characteristics of the constraints and the relative sizes of the set of test cases that kill the mutant and the set described by the constraint system. The probability of choosing at least one effective test case over n trials is equivalent to the probability of not failing on every choice. Since the probability of failing over n independent choices is $(1 - \rho)^n$, the probability of at least one success over n choices is given by

$$\Gamma = 1 - (1 - \rho)^n.$$

This is a function with a very fast growth rate as n gets large:

$$\begin{aligned}\Gamma &= 1 - \lim_{n \to \infty} (1 - \rho)^n \\ &= 1 - \lim_{n \to \infty} \left(1 + \frac{1}{n} * (-n * \rho)\right)^n \\ &= 1 - exp(-n * \rho)\end{aligned}$$

As long as ρ is not vanishingly small, Γ will approach 1. Not only that, but for reasonable values of ρ, Γ grows very quickly for small n. The lowest precision value in Table II is 0.20 for PAT. Even with this low value, it only takes 15 independent trials to achieve over a 95% probability of generating an effective test case. This means that we can increase the effectiveness of our test case set by generating several test cases for each constraint system. Since we create constraint systems conservatively, they never preclude test cases that will kill the mutants, but include some test cases that may not work. That is, a constraint system always describes a space that includes at least all the test cases that will kill the mutant.

Of course, adding test cases makes the testing process more expensive by increasing the overhead of generating and managing the test cases, but by automatically eliminating ineffective test cases we can avoid increasing the human time. Thus, even though the above formula cannot bound the number of trials needed (i.e., we cannot easily estimate n), we can keep generating test cases until we kill most of the mutants. Most importantly, we can expect this approach to increase the effectiveness of the test case set.

4.3 Adequacy Comparisons

Another important measurement of any test data generation technique is how the test data compares with data generated by other techniques. Comparisons of this nature are disappointingly rare in the literature. There seem to be two difficulties with performing these experiments; one is the lack of automated tools to implement the known testing techniques, and the other is the question of how to compare the testing techniques.

Measuring the relative adequacy, through the mutation score of a set of test data, is a comparison method that has been used in recent studies [11, 16, 27]. Relative adequacy not only measures the fault-detecting capabilities of the test data but also gives an indication of how well the data will test the software [4]. An experiment that used an earlier mutation system to measure the adequacy of several test data generation techniques was presented by DeMillo et al. [11]. They used a version of the TRITYP program that differed slightly from that used in the other experiments in this paper. Since no automated tools were available for the techniques, test data was generated by hand to satisfy the five techniques. We repeated this experiment using the same test data with the Mothra mutation system and the same version of TRITYP used in the previous study.

Table III gives the mutation scores of the test data on the Mothra system, including results from Godzilla-generated test data. Godzilla produced results that are closer to relative adequacy than the other five techniques. Of course, one could argue that in comparison with domain analysis (the most effective of the other five techniques), we added 461 test cases to kill 16 mutants. However, only 46 of the 536 constraint-based test cases were effective, compared with 41 from the domain-testing technique. As we pointed out before, these other 490 test cases can be generated and eliminated entirely automatically, without intervention from the tester.

Though time statistics were not kept for the original experiment, estimates by the participants indicate that it took approximately seven man-weeks to hand-generate and execute the data for the five methods. This is in sharp contrast to the CBT method, which took two hours and 20 minutes. Moreover,

Table III. Summary of Comparison Results

Technique	Test Cases	Killed	Mutation Score
Statement Coverage	5	642	.667
Branch Coverage	9	749	.778
Specifications Analysis	13	780	.811
Minimized Domain Analysis	36	932	.969
Domain Analysis	75	943	.980
Constraint-Based Testing	536	959	.997

this time was almost entirely computer time—the tester simply pushed the "go" button and examined the expected output. This human versus machine time tradeoff is the subject of our next experiment.

4.4 Mutant Detection Time

The original goal of this research was to automate test data generation within a mutation system. To compare Godzilla with the previous method, which was manual selection of test data, we generated test data with the Godzilla system and executed the test data using Mothra, recording the machine time, human time, and mutation score. Next, we constructed test data by hand to reach the same mutation score and recorded the same data. Since one element of the time measurement is the tester, wall clock time was used, and the test cases were executed on an unloaded Microvax II.

Table IV presents the times to generate and execute both sets of test data. All times are in wall clock minutes. The "Human Time" column gives the number of minutes used by the tester to create the test data, and "Machine Time" is the number of minutes used by the computer to execute all test cases and, for Godzilla, to generate the test cases (although we did not separate generation time from execution time, the time to execute was several times longer). The "Oracle Time" is the time spent examining the expected output of the test cases. The "Number TCs" column gives the number of test cases generated for the programs and, for Godzilla, the number of effective test cases. The machine time measures the time spent executing *all* the test cases, while the oracle time only measures time spent examining the effective test cases. We define the *speedup* (S) of generating the data automatically as the total time used by manually generated test data divided by the total time used by Godzilla-generated test data, representing the amount of time saved by automatic generation.

The differences are dramatic. The time to set up to generate test data automatically is constant (indeed, almost negligible), whereas the human time spent analyzing the program's mutants to develop test data generally increased as the programs got larger and more complicated. Although the machine time used during automatic generation was much more than the time used during human generation, mutant execution was the dominating factor, rather than the time spent generating test cases. The most significant point of this table is that by automating test case generation, we shift the most expensive part of using mutation from test case generation to output examination and mutant execution. Although this study does not tell us conclusively how much human effort will be saved by automatically generat-

Table IV. Testing Times (in minutes)

	MANUAL				GODZILLA				SPEEDUP
	Number TCs	Human Time	Machine Time	Oracle Time	Number TCs	Human Time	Machine Time	Oracle Time	
BSEARCH	13	75	2	1	235/15	1	24	2	2.80
BUBBLE	9	30	1	1	314/8	1	26	1	1.11
DAYS	52	1503	24	26	2637/31	1	1223	32	1.26
EUCLID	8	70	2	25	163/4	1	9	1	8.82
FIND	25	1328	5	4	455/13	1	92	2	14.07
PAT	20	284	4	2	359/22	1	79	2	3.51
TRITYP	56	609	12	14	713/45	1	157	11	3.76

ing test data, we can certainly conclude that constraint-based testing saves significant time, effort, and expense.

4.5 Combination of Test Case Constraints

The last experiment we present is aimed at reducing the number of test cases created by Godzilla. Not only does Godzilla produce several times as many test cases as a human tester, but many of these test cases do not kill any mutants because their target mutant(s) are killed by other test cases. This implies that there is some overlap in the test cases. We measure this overlap by "combining" the test case constraint systems to eliminate redundant test cases before they are generated.

Godzilla stores a program's constraint systems in a table, and the constraint satisfier loops through the table to generate in text test cases for each constraint system. We modified Godzilla so that when a test case T is generated to satisfy a constraint system C_i, Godzilla evaluates all constraint systems C_j, $C_j > C_i$, on T, and if T also satisfies C_j, Godzilla will not generate another test case for C_j.

Table V shows the results of this "combined satisfaction" method of test case reduction. The first three columns repeat the previous results of generating test cases with no reduction. The next three columns are the number of test cases generated by combined satisfaction, the number of mutants the reduced test case set killed, and the resulting mutation score. The effort made to reduce the test case increased the generation time by only a small amount (less than 50%), but since the test case sets were so much smaller, the amount of time spent executing mutants was reduced by a factor of 10 or more. Since execution time is much greater than generation time, the savings by this approach is significant. The decrease in mutation score of the reduced set was not large (between 0 and 14), indicating that it is possible to trade off a large amount of processing time for a small amount of testing strength.

5. CONCLUSIONS

This work is the first attempt to automatically generate test data to satisfy mutation adequacy. Constraint-based testing solves a major problem of using mutation testing as a practical method for testing software, that of creating test data. The method for generating test data has been fully implemented and integrated with the Mothra testing system—a mutation-testing-based testing system for Fortran-77. The Godzilla implementation is

Table V. Combined Constraints Results

	Full			Combine		
	TCs	Killed	MS	TCs	Killed	MS
BSEARCH	235	260	.99	11	227	.87
BUBBLE	314	303	1.00	7	303	1.00
DAYS	2637	2783	.97	255	2613	.91
EUCLID	163	171	1.00	20	171	1 00
FIND	455	934	1.00	30	934	1 00
PAT	359	445	1.00	30	381	86
TRITYP	713	822	.98	48	759	90

composed of over 15,000 lines of C code that includes the ability to create necessity constraints, derive reachability constraints for Fortran-77 programs, and satisfy the constraint systems to generate test cases for the test program.

Through its mutation-testing basis, CBT integrates the test data generation capabilities of path coverage techniques with the fault detection capabilities of mutation testing and subsumes techniques such as statement coverage, branch analysis, and domain analysis. The experiments described in this paper verify that Godzilla creates test cases that score well on the mutation system. Moreover, because the necessity constraints are derived from rules that describe test cases, the technique can easily be extended to handle other types of faults and to include other types of testing. These experiments also show that this method is a much more cost-effective way to create mutation-adequate test sets than the current manual method.

Perhaps the most useful result of this study is that an automatic test data generation capability allows us to view test cases as "throw-away" items rather than as expensive, scarce resources. With this view, we can generate test cases, toss them at mutants, and then throw them away if they do not work. Before this experiment, we had never considered examining the expected output of a test case **after** mutants had been executed. By postponing this expensive (manual!) step, generating test data automatically adds very little expense to the mutation process and removes a lot. In the future, we expect the mutation process to proceed quite differently from before. Initially, Godzilla will be used to generate a set of test cases (perhaps a test that is smaller than ultimately desired, using something similar to the combine option), and those test cases will be executed against the mutants. This process will be repeated, each time generating test cases to only target live mutants, until the test cases kill no more mutants. Then all ineffective test cases will be eliminated. Up to this point, the process has been entirely automatic. To finish testing, the tester will examine expected output of the effective test cases and either add test cases to kill the remaining mutants or decide that they are equivalent.

ACKNOWLEDGMENTS

We would like to thank the referees for clearly expressing many inaccuracies and misunderstandings due to the wording of our text and for innumerable rewording suggestions.

REFERENCES

1. BALDWIN, D., AND SAYWARD, F Heuristics for determining equivalence of program mutations. Res. Rep. 276, Dept of Computer Science, Yale Univ., New Haven, Conn., 1979.
2. BUDD, T. A. Mutation analysis of program test data Ph.D. dissertation, Yale Univ., New Haven, Conn., 1980.
3. BUDD, T A., AND ANGLUIN, D Two notions of correctness and their relation to testing. *Acta Inf. 18*, 1 (Nov 1982), 31–45.
4. BUDD, T., DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 1980 ACM Principles of Programming Languages Symposium* ACM, New York, 1980, 220–233.
5. CLARKE, L. A. A system to generate test data and symbolically execute programs *IEEE Trans Softw. Eng. 2*, 3 (Sept. 1976), 215–222.
6. CLARKE, L. A., AND RICHARDSON, D. J. The application of error-sensitive testing strategies to debugging. In *Symposium on High-Level Debugging* ACM SIGSOFT/SIGPLAN, New York, 1983, 45–52.
7. CLARKE, L. A., AND RICHARDSON, D. J. Applications of symbolic evaluation. *J. Syst. Softw. 5*, 1 (Jan. 1985), 15–35.
8. CRAFT, W. M Detecting equivalent mutants using compiler optimization techniques. Master's thesis, Dept. of Computer Science, Clemson Univ., Clemson, S. Carol., 1989.
9. DEMILLO, R. A., AND OFFUTT, A. J. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng. 17*, 9 (Sept. 1991), 900–910
10. DEMILLO, R. A., GUINDI, D S., KING, K. N., MCCRACKEN, W. M., AND OFFUTT, A. J. An extended overview of the Mothra software testing environment. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis* (Banff, Alberta, July 1988). IEEE Computer Society Press, Los Alamitos, Calif., 142–151.
11. DEMILLO, R. A, HOCKING, D E., AND MERRITT, M. J. A comparison of some reliable test data generation procedures. Tech. Rep. GIT-ICS-81/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, 1981
12. DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Comput. 11*, 4 (Apr. 1978), 34–41.
13. FOSDICK, L. D., AND OSTERWEIL, L. J. Data flow analysis in software reliability. *ACM Comput. Surv. 8*, 3 (Sept. 1976), 305–330
14 GELLER, M. Test data as an aid in proving program correctness. *Commun. ACM 21*, 5 (May 1978), 368–375.
15. GIRGIS, M. R., AND WOODWARD, M. R. An integrated system for program testing using weak mutation and data flow analysis. In *Proceedings of the 8th International Conference on Software Engineering* (London, UK, Aug., 1985). IEEE Computer Society, Los Alamitos, Calif., 313–319.
16. GIRGIS, M. R., AND WOODWARD, M R. An experimental comparison of the error exposing ability of program testing criteria. In *Proceedings of the Workshop on Software Testing*. IEEE Computer Society Press, Los Alamitos, Calif., 61–73.
17. GOSLING, J. Algebraic constraints. Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Penn., 1983.
18. GOURLAY, J. S. A mathematical framework for the investigation of testing. *IEEE Trans. Softw. Eng. 9*, 6 (Nov. 1983), 686–709.
19. HOARE, C. A. R. Proof of a program: Find. *Commun. ACM 14*, 1 (Jan. 1971), 39–45.
20. HOWDEN, W. E. Weak mutation testing and completeness of test sets. *IEEE Trans Softw Eng. 8*, 4 (July 1982), 371–379.
21. KING, K. N., AND OFFUTT, A. J. A Fortran language system for mutation-based software testing. *Softw. Pract. Exper. 21*, 7 (July 1991), 685–718.
22. LASKI, J. On data flow guided program testing. *Sigplan Not. 17*, 9 (Sept. 1982).
23. MANNA, Z. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.
24. MARICK, B. The weak mutation hypothesis. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification* (Victoria, British Columbia, Canada, Oct. 1991). IEEE Computer Society Press, Los Alamitos, Calif., 190–199.

25 Morell, L. J. A theory of error-based testing. Ph.D. dissertation, Univ. of Maryland, College Park, Md., 1984.
26. Morell, L. J. A theory of fault-based testing. *IEEE Trans. Softw. Eng. 16*, 8 (Aug. 1990), 844–857.
27. Ntafos, S. C. An evaluation of required element testing strategies. In *Proceedings of the 7th International Conference on Software Engineering* (Orlando, Fla., Mar., 1984). IEEE Computer Society, Los Alamitos, Calif., 250–256.
28. Offutt, A. J. Automatic test data generation. Ph.D. dissertation, Georgia Institute of Technology, Atlanta, 1988.
29. Offutt, A. J. An integrated automatic test data generation system. *J. Syst. Integr. 1*, 3 (Nov. 1991), 391–409.
30. Offutt, A. J. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Meth. 1*, 1 (Jan. 1992), 3–18.
31. Offutt, A. J., and Lee, S. D. How strong is weak mutation. In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification* (Victoria, British Columbia, Canada, Oct., 1991). IEEE Computer Society Press, Los Alamitos, Calif., 200–213.
32. Offutt, A. J., and Pressley, D. L. A data flow oriented approach to the path expression constraint generation problem. Tech. Rep. 92-113, Dept. of Computer Science, Clemson Univ., Clemson, S. Carol., 1992.
33. Purdom, P., and Brown, C. *The Analysis of Algorithms*. Holt, Reinhart and Winston, New York, 1985.
34. Ramamoorthy, C. V., Ho, S. F., and Chen, W. T. On the automated generation of program test data. *IEEE Trans. Softw. Eng. 2*, 4 (Dec. 1976), 293–300.
35. Rapps, S., and Weyuker, E. J. Data flow analysis techniques for test data selection. In *Software Engineering 6th International Conference*. IEEE Computer Society Press, Los Alamitos, Calif., 1982.
36. Richardson, D. J., and Thompson, M. C. The relay model for error detection and its application. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis* (Banff, Alberta, Canada, July 1988). IEEE Computer Society Press, Los Alamitos, Calif., 223–230.
37. Weyuker, E J. Assessing test data adequacy through program inference. *ACM Trans. Program. Lang. Syst. 5*, 4 (Oct. 1983), 641–655.

Received October 1990; revised January and July 1992; accepted July 1992