Constraint-Based Automatic Test Data Generation

Richard A. DeMillo and A. Jefferson Offutt

Abstract—This paper presents a new technique for automatically generating test data. The technique is based on mutation analysis and creates test data that approximates relative adequacy. The technique is a fault-based technique that uses algebraic constraints to describe test cases designed to find particular types of faults. A set of tools, collectively called Godzilla, has been implemented that automatically generates constraints and solves them to create test cases for unit and module testing. Godzilla has been integrated with the Mothra testing system and has been used as an effective way to generate test data that kill program mutants. The paper includes an initial list of constraints and discusses some of the problems that have been solved to develop the complete implementation of the technique.

Index Terms—Mothra, constraints, fault-based testing, mutation analysis, software testing, test data generation.

I. INTRODUCTION

SOFTWARE testing is the primary method by which testers establish confidence in the correctness of software. This confidence is ordinarily established by executing the software on test data chosen by some systematic testing procedure. In recent years, the phrase "fault-based testing" has been applied to techniques that choose test data that attempt to show the presence (or absence) of specific faults during unit testing. Techniques have been developed that determine whether test data can detect specific faults (i.e., mutation analysis [12]), and the theoretical properties of fault-based testing have been explored [5], [27]. One of the most difficult and expensive parts of applying these techniques has been the actual generation of test data—which has traditionally been done by hand. The work described in this paper represents what we believe to be the first attempt to automatically generate test data that meet the fault-based testing criteria.

This technique is expected to be applied in a structured way. In the worst case, the algorithms described below use space proportional to the square of the number of variable references. The underlying algorithms are NP complete or worse. In order to make the algorithms feasible, a divide-and-conquer strategy based on program structure is required. When applied during unit testing, the cost of test data generation is reasonable. Tests of large systems can be composed from tests of smaller subsystems and functions if the system structure admits, for example a hierarchical decomposition. Tests of large, monolithic software systems are probably intractable.

We consider a program unit to be a subroutine, function, or (relatively) small collection of related subroutines and functions. Throughout this paper we use the term program interchangeably with program unit.

We begin by highlighting some of the major theoretical results in software testing to put the current work in perspective. Except for this introductory section, the discussion is informal and we present results in an intuitive, descriptive manner. This paper describes a testing technique that is practical, general-purpose, and, most importantly, has been completely automated.

A. Background

We use the following notation and assumptions. First, we assume that all programs satisfy functional specifications. If P is a program, then $P(t)$ denotes the value of the function computed by P at the point t. The set of all points at which P defines a definite value is the domain of P. The letter D is usually used to denote the domain of a program. Much of the previous work in software testing has been based on test case *reliability* as defined by Howden:

Definition 1 [20]: If P is a program to implement function F on domain D, then a test set $T \subset D$ is *reliable* for P and F if $\forall\ t \in T,\ P(t) = F(t) \Rightarrow \forall\ t \in D,\ P(t) = F(t)$.

That is, successful execution of a reliable test set implies that the program is correct on all inputs. Although Howden went on to show that finite reliable test sets exist [20], he also showed that there is no effective procedure for generating reliable test sets of *finite* size. An alternative testing criteria that has been proposed is that of test case *adequacy*:

Definition 2 [5], [12]: If P is a program to implement function F on domain D, then a test set $T \subset D$ is *adequate* for P and F if $\forall$ programs Q, if $Q(D) \neq F(D) \Rightarrow \exists t \in T$ such that $Q(t) \neq F(t)$.

That is, a test set is adequate if it causes all incorrect versions of the program to fail to execute successfully. Adequacy requires that our test set detects faults rather than show correctness, making test data adequacy a more intuitively appealing definition. Unfortunately, Budd and Angluin [5] show that there is no effective procedure for either generating adequate test sets or for detecting that a given test set is adequate. The practical problem with adequacy is that we are, in effect, asking that test cases distinguish a correct program from all possible incorrect programs, which is of course undecidable.

DeMillo *et al.* [12] proposed reducing the number of incorrect programs considered by distinguishing correct programs from programs that are "almost" correct. This approach is called *relative adequacy* or, more commonly, *mutation adequacy*:

Definition 3 [12]: If P is a program to implement function F on domain D, and Φ is a finite collection of programs, then a test set $T \subset D$ is adequate for P *relative* to Φ if $\forall$ programs $Q \in \Phi$, if $Q(D) \neq F(D) \Rightarrow \exists t \in T$ such that $Q(t) \neq F(t)$.

In other words, a relative adequate test set causes a specific, finite number of incorrect versions of the program to fail. An intuitive rationale for this restriction is the "competent programmer assumption" [12]. The competent programmer assumption states that programmers do not necessarily produce programs that are correct, they create programs with bugs. Although such programs may fail on certain inputs, they will work correctly for most. In their paper, Budd and Angluin [5] define two related terms and explore their theoretical distinctions:

Definition 4 [5]: A procedure $A(P, T)$ is an *acceptor* if it returns TRUE if T is adequate for P relative to the set Φ, and FALSE otherwise.

Definition 5 [5]: A procedure $G(P)$ is a *generator* if it produces a test set T that is adequate for P relative to Φ.

Budd and Angluin show that computable acceptors and generators for adequate test sets do not necessarily exist, and even more discouraging, that *finite* adequate test sets do not always exist. On the other hand, finite *relative* adequate test sets do exist. Unfortunately, we cannot always find procedures for generating and accepting relative adequate test sets.

The above results are useful in a variety of ways. Most importantly, they tell us what we can and cannot gain from testing. Specifically, we cannot in general show correctness through testing, neither can we generate reliable or adequate test sets. On the other hand, relative adequate test sets do exist, and in this paper we present a technique to generate test sets which 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 based techniques, the tester is left wondering whether the program is correct or if there are faults that have not yet been discovered. Adequacy-based 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 theoretical way to weaken reliability.

B. Generating Adequate Test Sets

In the past decade, several acceptors for relative-adequate test sets have been implemented in the form of mutation analysis systems. Early systems included PIMS [12] and EXPER [1]. The most recent mutation analysis system is the Mothra testing system [10]. These systems have all created the incorrect programs in Φ by applying *mutation operators* to the test program. Each program in Φ is referred to as a *mutant* of the original program. Mutation systems carry out an interactive dialogue with the tester to find test data that executes correctly on the original program and that causes each mutant program to fail. Test data that causes a mutant program to fail is said to *kill* the mutant.

Generating relative-adequate test sets is a very labor-intensive task. To generate relative-adequate test sets, a tester has been forced to interact with a mutation system by exhaustively examining the remaining live mutants and hand-designing tests to kill them. In this paper we describe an automated technique for producing test data. Specifically, we describe a technique for producing test data that approximates relative-adequacy—in Budd and Angluin's terms, a generator. The technique has been fully implemented in a tool called the Godzilla Test Data Generator, which is integrated with the Mothra software testing system.

Our strategy is to represent the conditions under which mutant programs will die as simple algebraic constraints on the tests, and to automatically generate data to satisfy the constraints. We call this approach *constraint-based testing* (CBT). CBT is an approximation technique. Godzilla generates test data that is approximately relative-adequate, and all claims in this paper are meant to be taken with this approximation in mind. By using Godzilla's test data in a mutation system, however, we can measure the extent of the approximation by the number of mutants killed. This will tell us exactly how close the test data is to being relative adequate.

The work described here is part of the Mothra Software Test Environment Project [15]. We have implemented constraint-based testing in a tool called Godzilla, which is integrated with Version 1.2 of the Mothra Software Testing environment. Fig. 1 shows the basic menu structure of Godzilla as used in Mothra. Throughout this paper we elaborate on the functional capabilities of Godzilla that are shown in Fig. 1. At the same time, we present a rationale for CBT in terms of mutation analysis, since the two methods share a common underlying theoretical basis.

Mothra is the first product of a multi-institutional software testing research group. The ultimate goal of this research group is to completely automate the testing and measurement process so that when a correct program is submitted for testing, an adequate test set is produced as evidence of correctness without human intervention [11]. In this ideal system, when an incorrect program is submitted to the test environment, the tester would be provided—again, without human intervention—tests on which the program fails and a list of possible faults and their locations in forms suitable for input to a debugging tool. Although this goal may not be realizable, a system that is as close to this goal as possible could significantly improve the state of practice in software testing.

C. Overview of Mutation Testing

Mutation analysis is a fault-based testing method that measures the adequacy of a set of externally created test cases [12], [14], [15]. In practice, a tester interacts with an automated *mutation system* to determine and improve the adequacy of a test data set. This is done by forcing the tester to test for specific types of faults. These faults are represented as simple syntactic changes to the test program[1] that produce mutant

---

[1] Since software testing is often concerned with arbitrary subprograms as well as main programs, we use the term "program" to include programs, subroutines, and functions.
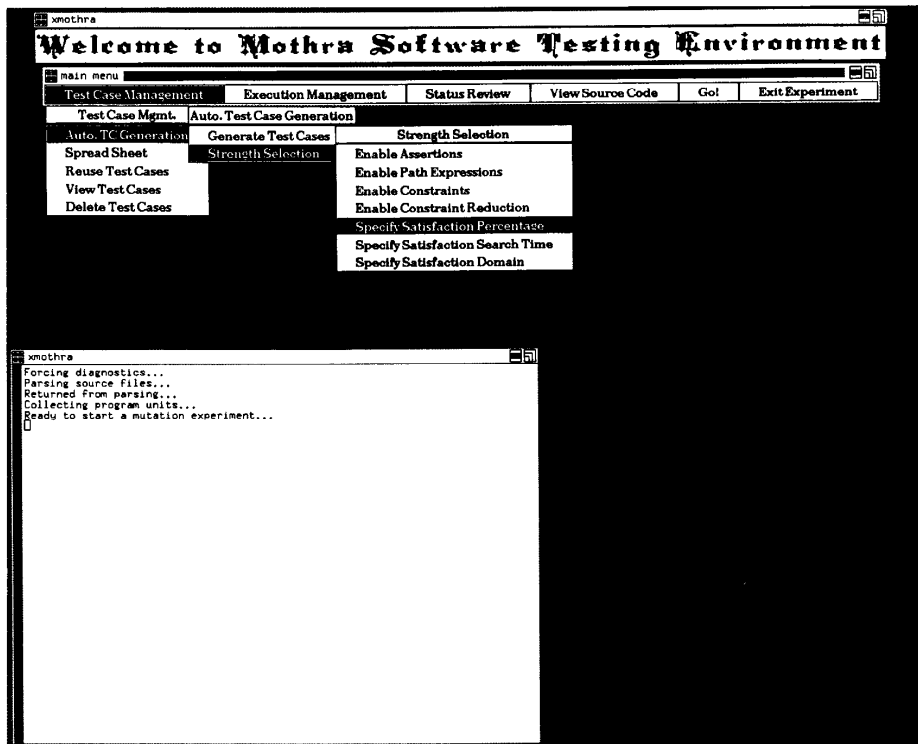
Fig. 1. X interface to Mothra and Godzilla.

FUNCTION MAX (M,N)	
1	MAX = M
Δ	MAX = N
Δ	MAX = **ABS** (M)
2	IF (N .GT. M) MAX = N
Δ	IF (N .LT. M) MAX = N
Δ	IF (N .GE. M) MAX = N
3	RETURN

Fig. 2. Function MAX.

programs. For example, consider the function **MAX** shown in Fig. 2 that returns the larger of its two integer inputs. The lines preceded by a "Δ" represent mutants of the corresponding line in the test program. Note that each of the four mutants in Fig. 2 represents a separate program.

Mothra uses 22 mutation operators for testing Fortran programs, and over 70 for testing C programs. These operators are designed to represent common mistakes that a programmer might make. These particular mutation operators [25] represent more than 10 years of refinement through several mutation systems. These operators explicitly require that the test data meet statement and branch coverage criteria, extremal values criteria, domain perturbation, and that they also directly model many types of faults. The goal of the tester during mutation analysis is to create test cases that differentiate each mutant program from the original program by causing the mutant to produce different output. When the output of a mutant program differs from the original program on some input, that mutant is considered *dead* and is not executed against subsequent test cases. A test case set that kills all mutants is adequate relative to those mutants. This allows mutation analysis to indicate how well the program has been tested though an adequacy score, or *mutation score* (formally defined in Section V).

The ideas behind mutation analysis may be clarified by looking at the process followed during mutation analysis. Upon adding a new test case to the mutation system, the test case is first executed against the original version of the test program to generate the *expected output* for that test case. An oracle (usually a human tester) is then asked to examine the output; if the output is incorrect, the program should be corrected and the mutation process restarted; otherwise each live mutant is executed against the new test case. The output of the mutant program is compared to the expected output; mutants are killed if their output does not match that of the original, and remain alive if they do.

After all mutants have been executed, the tester is left with two kinds of information. The proportion of the mutants that die indicates how well the program has been tested. The live mutants indicate inadequacies in the current test set (and potential faults in the program). The tester must add additional test cases to kill the remaining live mutants. The process of adding test cases, examining expected output, and executing mutants continues until the tester is satisfied with the number of dead mutants.

The expected output examination is an important step in this process that is often lost in descriptions of mutation testing. This is the point at which the tester finds errors during mutation testing. If a fault exists in the test program there will usually be one or more mutants that can only be killed by a test case that also causes the fault to result in a failure. Oftentimes the faults which are revealed in this way are quite complex and are related in subtle ways to the mutant that the tester was trying to kill.

It is generally impossible to kill all mutants of a program because some changes have no effect on the functional behavior of the program. The last mutant shown of **MAX** is a mutant that is functionally *equivalent* to the original program. Since equivalent mutants always produce the same output as the original program, they can never be killed. In current mutation systems, equivalent mutants are recognized by human examination or by relatively primitive heuristics [2], [9]. In fact, a complete solution to the equivalence problem is not possible [5]. Recognizing equivalent mutants and creating test cases have been the two most human-intensive and therefore the most expensive actions within previous mutation systems. Reducing the amount of human interaction necessary to perform mutation is a major goal of constraint-based testing.

II. KILLING MUTANTS

Generating test data with or without a mutation system is a difficult task that poses complex problems. Practical test-data-generation techniques attempt to choose a subset of the input space according to some testing criteria. The assumption behind any criteria for generating test data is that the subset of inputs chosen will find a large portion of the faults in the program as well as help the tester establish some confidence in the software. Test cases that score well on a mutation system will have just those properties, because the test data is close to being adequate relative to the mutation operators. Specifically, the test data will find most, if not all, faults represented by the mutation operators.

Constraint-based testing uses the concepts of mutation analysis to automatically create test data. This test data is designed specifically to kill the mutants of the test program. Such test data can be used to kill mutants within a mutation system, or it can be used independently as highly effective test cases. Moreover, because of the mutation analysis basis, the test data is guaranteed to have such properties as containing extremal values [29], covering all statements [6], [7], and all branches [7]. These results are presented by Acree *et al.* [1].

The concept of using mutations to generate specific tests was also suggested by Howden [21] in his "weak mutation" approach and by Budd [4] in his dissertation. The present work, however, represents what is apparently the first general and implemented attempt to generate relative-adequate test sets.

Informally speaking, the method works as follows. 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 each 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, attempts to select effective test cases from the total domain of inputs for a program. One way to automatically generate test cases to kill mutants is to filter out the ineffective test cases. Such a filter may be described by mathematical constraints. In this way, generating mutation-adequate test sets is reduced to solving algebraic problems.

III. ADEQUACY-BASED TEST CASE CONSTRAINTS

A test case filter that generates test cases to kill mutants needs to have one simple, broad characteristic: **it must generate test data that makes a difference in the mutant's behavior**. That is, since the mutant is represented by a single change to the source program, the execution state of the mutant program must differ from that of the original program after some execution of the mutated statement. We state this characteristic as the *necessity condition*:

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

To see why, simply note that since M is syntactically equal to P except for the mutated statement S, if the states of the two programs do not differ after the last execution of S, they will never differ. The state difference that we wish to cause is described in terms of the same program symbols that describe mutation operators and depends directly on those operators. We have defined a list of *necessity conditions* that a test case must meet to kill each type of mutation defined in the Mothra testing system.

We use the term "necessity condition" because, although it is clear that an incorrect state is necessary, it cannot be guaranteed to be sufficient to kill the mutant. For a test case to kill a mutant, it must create incorrect output, in which case the final state of the mutant program differs from that of the original program. Although deriving and finding test cases that meet this *sufficiency condition* is certainly desirable, it is impractical in practice. Completely determining the sufficiency condition implies knowing in advance the path that a program will take, which is, of course, intractable.

The sufficiency problem has been considered by other researchers. Howden has studied it from the mutation analysis (or acceptor) point of view in his weak mutation approach [21]. He proposed eliminating much of the execution expense incurred during mutation analysis by comparing the states of the mutant and original program immediately after execution of the mutated statement, rather than executing the mutant program to completion. At this point, one of two situations can occur:

1) If the states of the two programs are the same, then the final states will also be the same and the mutant will not be killed by the test case

2) If the states are different, then weak mutation cannot guarantee either that the mutant will live or die.

Howden's original proposal was that in the second case we would execute the mutant program to completion to be certain that the test case killed the mutant. We can also take a more liberal approach in this case by assuming that the mutant is dead. Of course, this assumption will sometimes give incorrect results (we will mark mutants dead that should not be dead), but could be close enough to have minimal impact on the overall effectiveness of the test data. Since weak mutation has never been implemented or experimented with, there is no empirical basis for interpreting weak mutation scores. We are currently implementing a weak mutation system to answer these questions.

Budd and Angluin [5] also discussed the sufficiency problem, using the term *coincidental correctness*. In their terms, an incorrect program appears to be coincidentally correct if a test case causes the program to have an intermediate erroneous state but the program still produces correct output. Morell [28] has also explored this problem in discussing the *extent* of fault-based testing techniques. A *global extent* testing technique requires that the effects of a program fault cause a failure in the output, or final program state. A *local extent* testing technique only requires that the effects of a program fault result in an incorrect local, or intermediate, program state. Thus global extent techniques must solve the sufficiency problem. Morell points out that although global extent techniques are more general and more desirable, local extent techniques are more practical.

CBT's necessity condition is similar to Morell's *creation condition* and the sufficiency condition is similar to his *propagation condition* [27], [28]. Whereas Morell's creation condition describes program states in which mutants would alter the state of the program, the necessity condition describes a test case that will cause an incorrect state in **one** mutant of the program. Richardson and Thompson [34] extended Morell's ideas using a path-analysis-based approach. In their terms, a *potential fault* is a discrepancy between the program being tested and a hypothetically correct version of the program (these are modeled as mutants in mutation analysis). A potential fault *originates* if the smallest expression containing the potential fault evaluates incorrectly. The potential fault *transfers* to a "super-expression" which 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 cause incorrect output from the program. This technique seems similar in intent to CBT, but differs in method. CBT is based explicitly on mutation and the method for generating test cases focuses on killing mutants.

Our system currently assumes that if a test case meets the necessity condition, it will usually meet the sufficiency condition. If not, the output of M is identical to that of P and the mutant does not die. Since the test case meets the necessity condition, the state of the mutant program M diverges from the original program P at the point following the mutated statement. So the state of M first diverges from P, then returns to that of P.

This divergence and subsequent convergence in the states of two programs can happen in one of three ways. First, the test case can be weak—although it had an effect, it did not

FORTRAN SOURCE	CONSTRAINT
FUNCTION MAX (M,N)	
1 MAX = M	
Δ MAX = N	$M \neq N$
Δ MAX = ABS (M)	$M < 0$
2 IF (N .GT. M) MAX = N	
Δ IF (N .LT. M) MAX = N	$(N > M) \neq (N < M)$
Δ IF (N .GE. M) MAX = N	$(N > M) \neq (N \geq M)$
3 RETURN	

Fig. 3. Constraints of function MAX.

change the state in a way that results in a change in the final output. Second, the program can be robust enough to recognize the intermediate state as being erroneous and return to a correct state. Third, the intermediate state that the mutant affected can 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 case seems to be rare—that is, randomly chosen test cases exhibiting this property occur with relatively low probability. This has been observed elsewhere [13], [30], is indirectly supported by recent theoretical results [19], and we speculate that it is a useful working assumption in practice. The second case corresponds to a kind of fault tolerance in the tested software; this case is interesting enough to warrant further investigation on its own merits. In the absence of explicit fault tolerance, however, such robustness is probably rare. The third, of course, provides an opportunity to detect equivalent mutants, a difficult problem in its own right.

A. Necessity Constraints

Since mutation operators represent syntactic changes, a test case that meets a necessity condition must ensure that the syntactic change effected by the mutation results in a state difference for the program. As an example, Fig. 3 shows the function **MAX** with the mutants from Fig. 2 and the necessity constraints for the four mutants.

The first constraint in Fig. 3 can be generalized to the form $X \neq Y$, where X and Y represent arbitrary scalar variables. In this light, we have defined a necessity constraint *template* for each mutation type. The templates used by Godzilla are presented in Table I and are taken directly from the mutation operators used in Mothra [1], [25]. The symbols X and Y are used for scalar variables, A and B for array names, e_1 and e_2 for arbitrary expressions, ρ and ϕ for binary operations, and C for a constant.

Many of the mutation types used by Mothra cause one program symbol to be replaced by another program symbol. The necessity constraint templates for mutation that cause replacements require that the value of the original program symbol differ from the value of the replaced program symbol. The first constraint in Fig. 3 is an example of this type of mutation operator and its constraint.

A mutation type that does not cause program symbol replacements is the *abs* mutation type. The *abs* mutation actually involves three separate mutations applied to expressions: insertion of an absolute value operator; insertion of a NEGABS

TABLE I
CONSTRAINT TEMPLATES

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

TABLE II
MUTATION TYPES WITH NO TEMPLATES

Type	Description
crp	constant replacement
dsa	data statement alterations
glr	GOTO label replacement
rsr	RETURN statement replacement
san	statement analysis
sdl	statement deletion
src	source constant replacement
uoi	unary operator insertion

equivalent. We are currently exploring ways of combining data flow analysis with algebraic transformations to automatically detect these types of equivalent mutants.

Several Fortran mutation types used in Mothra that are not represented in Table I are shown in Table II. These mutation types represent changes that will, by definition, always satisfy the necessity condition. For example, the *crp* mutation replaces one constant with another one, and the obvious necessity constraint, that the two constants have different values, is independent of the test case. In fact, the Mothra system does not generate these mutants unless the constant values differ. Likewise, *dsa* mutations, *src* mutations, and *uoi* mutations all change the value of constants and the necessity condition is automatically satisfied. The other mutation types, *glr*, *rsr*, *san*, and *sdl* mutations change the state of the mutant program by changing the flow of control. Thus they too automatically satisfy the necessity condition. Experimentation has shown that mutants of these types are almost invariably killed by test cases generated for other mutants [30].

B. Predicate Constraints

Our second approximation for satisfying the sufficiency condition is to extend the necessity conditions to cause differences in program predicates. Early experimentation with the necessity constraints showed that the test cases generated were relatively ineffective at killing mutants that involved changes to predicate expressions. For example, for one program (the **TRITYP** program shown in Section V, Fig. 6), over 60% of the constraints involved predicates. While the test data generated for the constraints that did not involve predicates killed around 90% of the mutants, the test data for the constraints that did involve predicates only killed 65% of the mutants. Although the test cases being generated caused an immediate effect on the state of the mutant program, the (Boolean) result of the mutated predicate often was the same as that of the original program. An example of this problem is shown in Fig. 4, where a constant for scalar variable replacement operator (*scr*) replaces the variable "I" with the constant "3." Although the test case in Fig. 4 satisfies the necessity constraint $I \neq 3$, the result of the predicate test remains unchanged: $I + K = 14$ and $3 + K = 10$, both of which are greater than J.

To overcome this difficulty, we extended the necessity constraints to force mutated predicates to have different values than the corresponding predicates in the original program. The technique to ensure a predicate difference is as follows. For each mutation on an expression e in the program, we construct the mutated expression e'. From these two expressions, we

operator, which takes the negation of the absolute value; and the insertion of the ZPUSH operator. The ZPUSH mutant dies immediately if its argument is zero[2]. In order to kill these mutants we need the expression to successively take on a negative value, a positive value, and the value zero. The *abs* mutation operator is included precisely to guide the tester to follow the heuristic of testing each expression with zero, a negative value, and a positive value.

The binary operator replacement mutations (*aor*, *lcr*, *ror*) are also worth mentioning. To kill one of these mutants, the result of the entire mutated expression must differ from the original expression's result. The *aor* operator in particular involves four separate mutations. In its basic form, an *aor* mutation replaces one arithmetic operator with another (in Table I, ρ is replaced by ϕ). The necessity constraint requires that the values of the two expressions differ. In the *aor* operator's next two versions, the expression is modified to return just the left-hand side of the expression, and just the right-hand side of the expression. Again, the necessity constraint requires that the value of the original expression differs from that of just the right-hand side and then just the left-hand side. In the *aor* operator's final version, the *Mod* operator returns the remainder of e_1 divided by e_2.

The last mutant in Fig. 3 is a relational operator mutant (*ror*), and is equivalent. The constraint given, $(N > M) \neq (N \geq M)$, can be reduced to $(N = M)$. A simple application of data flow analysis shows that in this case, the variable MAX will always have the same value—proving that the mutation is

---

[2]ZPUSH requires test data to force all expressions to have the value of zero, a common testing heuristic.

Original statement	IF (I+K .GE. J) THEN
Mutated statement	IF (3+K .GE. J) THEN
Necessity constraint	$I \neq 3$
Test case	I = 7, J = 9, K = 7

Fig. 4. Predicate problem.

construct the *predicate constraint* as $e \neq e'$. For the example above, the predicate constraint is:

$$(I + K \geq J) \neq (3 + K \geq J).$$

The test case in Fig. 4 substitutes into this predicate constraint as $((7 + 7 \geq 9) \neq (3 + 7 \geq 9)) - FALSE$, not satisfying the predicate constraint. The test case that Godzilla generated to satisfy this constraint is 7, 10, 7, in which case $I + K = 14$ and $3 + K = 10$. This made $((7 + 7 \geq 10) \neq (3 + 7 \geq 10)) = TRUE$. For the **TRITYP** program, the percent of mutants killed increased from 74% without predicate constraints to 88% with predicate constraints.

The constraint-based technique is apparently robust enough to allow many other kinds of constraints to be added. Indeed, we expect to add other types of constraints, further refining or adding more fault-detection power to the test cases. For example, it may be possible to include domain testing [35] or other path-based testing techniques as additional constraints. Moreover, we are continuing to strengthen the necessity constraints.

IV. CONSTRAINT SATISFACTION

Although mathematical constraints have been proposed by a number of researchers as a method for describing test cases [6], [22]–[24], [33], many current systems work by randomly generating values until the criteria is satisfied [16], [18]. Finding values to satisfy constraints is a difficult problem that arises in such diverse areas as computability theory, operations research, and artificial intelligence [32]. Constraint satisfaction is known to be a good model of circuit test generation and analysis of line drawings as well as many combinatorial problems. The constraint satisfaction algorithms in Godzilla are the naive ones. It was therefore rather unexpected that they perform so well. Research is currently underway to develop more sophisticated approaches and to understand from a theoretical point of view the special characteristics of this version of the general constraint satisfaction problem. Since the sequel presents experimental results based on these algorithms, we present a brief summary of the algorithms for completeness.

Our approach employs a battery of procedures that work efficiently and produce satisfying test cases when the constraints have a reasonably simple form. These heuristics are briefly described here; the full procedure is presented in Offutt's dissertation [30]. These heuristics do not attempt to find an exact solution to the constraint satisfaction 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. Our heuristics meet this requirement.

Since constraints can be of arbitrary complexity, the problem is clearly as difficult as nonlinear programming [3]. In fact, constraint satisfaction is NP-complete even if the problem is restricted to three binary-valued variables [32]. Also, each constraint satisfaction problem is reducible to resolution theorem proving, which is exponential time in the worst case. On the other hand, there is much evidence that improvements of many orders of magnitude are possible, both in the general case and in specific cases. Fortunately, test case constraints tend to exhibit certain characteristics that allow us to, in most cases, make simplifications in the problem. A case study analysis of the 3600 test-case constraints generated for a group of Fortran programs [13] has shown that the constraints are almost always linear, generally involve only a few variables per constraint system, contain significantly more constant values than variables, and contain many more inequalities than equalities. These characteristics suggest that simple and fast procedures should be effective at satisfying the types of constraints created by a test-case generator.

A. Constraint Representation

The basic component of a constraint in Godzilla is an *algebraic expression*, which is composed of variables, parentheses, and programming language operators. Expressions are taken directly from the test program and come from right-hand sides of assignment statements, predicates within decision statements, etc. A *constraint* is a pair of algebraic expressions related by one of the conditional operators $\{>, <, =, \geq, \leq, \neq\}$. Constraints evaluate to one of the Boolean values TRUE or FALSE and can be modified by the negation operator NOT ($\neg$). A *clause* is a list of constraints connected by the logical operators AND ($\bigwedge$) and OR ($\bigvee$). A *conjunctive clause* uses only the logical AND, and a *disjunctive clause* uses only the logical OR. Godzilla keeps all constraints in *disjunctive normal form* (DNF), which is a list of conjunctive clauses connected by logical ORs.

In the following, we will refer to a *constraint system* as a DNF clause that represents one test case. DNF is used for convenience during constraint generation (each conjunctive clause within a path expression represents a unique path to a statement) and for ease of satisfaction (only one conjunctive clause needs to be satisfied).

B. Domain Reduction Constraint Satisfaction Procedure

Godzilla's *domain reduction* works by successively finding values for the variables in a constraint system that are consistent with remaining constraints. By the time we have to make arbitrary assignments of values to variables, we have already eliminated many possible incorrect values. We view the test case constraints as defining an N-dimensional space of input values, called the satisfying space. The procedure uses local information in the constraints to find values for variables, and then uses back-substitution to simplify the remaining constraints. Despite the fact that many algorithms for restricted versions of the problem use some form of domain reduction,
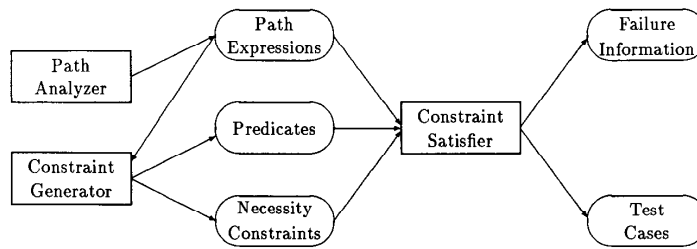
Fig. 5. Godzilla automatic generator architecture.

it is known to be of limited value in general constraint satisfaction [32]. For constraint-based testing, however, the situation is not so clear-cut; indeed, domain reduction is sometimes a very effective strategy.

Initially, each variable is assigned a domain that includes all possible values for a variable of that type. Each constraint within the system is viewed as a statement that reduces the domain of values for the variable(s) in the constraint. Constraints of the form $x \; \mathcal{R} \; c$, where x is a variable, c is a constant, and $\mathcal{R}$ is a relational operator, are used to reduce the current domain of values for x. Constraints of the form $x \; \mathcal{R} \; y$, where both x and y are variables, are used to reduce the domain of values for both x and y. This reduction uses substitution to eliminate expressions in the constraints, and also reduces the domain of values for individual variables. When no additional simplification can be done, a heuristic is employed to choose a value for one of the remaining variables in the constraints. The variable chosen is the variable with the smallest current domain. The value for this variable is chosen arbitrarily from its current domain of values, and is then back-substituted into the remaining constraints. This process is repeated until all variables have been assigned a value. By choosing a variable with a minimal current-domain size, we expect to have less chance of making a mistake by choosing a value that will cause a solvable constraint system to become unsolvable.

Each time a variable is assigned a value, its satisfying space is reduced by one dimension. As the number of dimensions is reduced, the constraints in the system become progressively simpler. Each variable assignment implicitly introduces a new constraint into the system of the form $(x = c)$, where c is a constant. If chosen poorly, this new constraint may make the region infeasible. The basic assumption is that because of the simple form of the test case constraints, these dimension-reducing constraints will not make the region infeasible. When they do, the infeasibility will often be easily recognized, since they show up as conflicting constraints such as $(x = 4 \land x > 8)$.

The domain-reduction procedure works well on constraints when a majority of the constraint clauses are of the form $x \; \mathcal{R} \; c$ or $x \; \mathcal{R} \; y$. In the test-case constraints studied elsewhere [13], 58% of the clauses were of the form $x \; \mathcal{R} \; c$, and 30% of the clauses were of the form $x \; \mathcal{R} \; y$. These observations were the prime motivations for choosing the heuristics we use in the domain-reduction procedure. The heuristic of evaluating sim-

ple constraints first allows known values to propagate through the constraints before variables need to be assigned arbitrarily. Moreover, by keeping track of the current domain of values for each variable, the procedure progressively encodes (and combines) constraints in a way that eliminates values that do not satisfy the constraints without eliminating values that do. By the time arbitrary decisions about a variable's value are made, many values that do not satisfy the constraints have already been eliminated—leaving a higher density of values that do satisfy them, which in turn lowers the probability for error.

Because of the heuristics used, the domain-reduction procedure works efficiently and accurately for constraint systems that largely contain clauses with a simple form. Moreover, when it is presented with complex constraints, the procedure may be slow and less accurate, but it does not give up.

V. EXPERIENCE WITH GODZILLA

We have implemented a system that uses CBT to generate test data for Mothra. Although the technique is language independent, the current tool generates test cases for Fortran 77 programs. Godzilla was implemented as separate tools in the programming language C on a DEC MicroVax II running the Berkeley Unix 4.3 operating system. Godzilla contains about 15 000 lines of source code and has been ported to a variety of machines, including Vaxes, a Pyramid, an AT&T 3b2 (running system V unix), and several Sun systems.

Fig. 5 highlights the major tools of Godzilla. The three major tools are the *Path Analyzer*, the *Constraint Generator*, and the *Constraint Satisfier*, represented in the figure by boxes. These tools, which are designed as separate entities and implemented as separate programs, communicate through files represented by ovals. The arrows indicate the flow of data through Godzilla.

For each statement in the original program, the path analyzer uses path-coverage techniques to create a *path expression* constraint such that if the test case reaches that statement, the constraint will be true. Ideally, we would like to create constraints for the inverse; if the constraint is satisfied, the statement will be executed. Creating these constraints, however, is intractable in general. In practical terms, however, satisfaction of the path expression constraints will guarantee execution of the targeted statement in the absence of

```
INTEGER FUNCTION TRITYP (I,J,K)

      INTEGER I,J,K
C     Return value is output from the routine:
C          TRITYP = 1 if triangle is scalene
C          TRITYP = 2 if triangle is isosceles
C          TRITYP = 3 if triangle is equilateral
C          TRITYP = 4 if not a triangle
C     After a quick confirmation that it is a legal
C     triangle, detect any sides of equal length.
      IF (I.LE.0.OR.J.LE.0.OR.K.LE.0) THEN
          TRITYP = 4
          RETURN
      ENDIF
      TRITYP = 0
      IF (I.EQ.J) TRITYP = TRITYP+1
      IF (I.EQ.K) TRITYP = TRITYP+2
      IF (J.EQ.K) TRITYP = TRITYP+3
      IF (TRITYP.EQ.0) THEN
C        Confirm it is a legal triangle before declaring
C        it to be scalene.
         IF (I+J.LE.K.OR.J+K.LE.I.OR.I+K.LE.J) THEN
```

```
                    TRITYP = 4
             ELSE
                    TRITYP = 1
             ENDIF
             RETURN
      ENDIF

C     Confirm it is a legal triangle before declaring
C     it to be isosceles or equilateral.
      IF (TRITYP.GT.3) THEN
             TRITYP = 3
      ELSE IF (TRITYP.EQ.1.AND.I+J.GT.K) THEN
             TRITYP = 2
      ELSE IF (TRITYP.EQ.2.AND.I+K.GT.J) THEN
             TRITYP = 2
      ELSE IF (TRITYP.EQ.3.AND.J+K.GT.I) THEN
             TRITYP = 2
      ELSE
             TRITYP = 4
      ENDIF
      END
```

Fig. 6. Subroutine TRITYP.

TABLE III
TEST CASE ADEQUACY RESULTS

	Size	TCs	M	D	E	MS	Time
BUBBLE	10	32	339	304	35	1.00	0:22
DAYS	28	419	3016	2624	139	.95	7:02
FIND	28	58	1029	953	75	.99	2:27
GCD	55	325	5063	4747	298	.99	14:24
TRITYP	27	420	970	862	107	.99	10:53

back branches[3]. In fact, the constraints are strong enough to ensure that all statements are executed in the presence of back branches that have a structured form (i.e., loops) [31].

The constraint generator constructs the necessity and predicate constraints described in Section III. Both the path expression and necessity constraints are stored in constraint tables and passed to the constraint satisfier. The satisfier gets each necessity constraint in turn, finds the corresponding statement, conjoins that constraint with the appropriate path expression, and uses the procedure outlined in Section IV to generate a test case to solve the conjunction. If a test case cannot be generated for some reason (usually because the satisfier failed), the information about the failure can be supplied to the tester. The failure information includes the constraints themselves, whether the constraint system was detectably unsolvable, and the parameters used by the satisfier.

A program that has been widely studied in the literature is the triangle classification program TRITYP [1], [8], [12], [33]. The Fortran version of TRITYP in Fig. 6 accepts three integers that represent the relative lengths of the sides of a triangle. The result is either a 1, 2, 3, or 4, indicating that the input triangle is equilateral, isosceles, scalene, or illegal, respectively. One important measurement of test data adequacy is the *mutation score*. For a program P and set of test data T, the number of mutants is M, the number of dead mutants is D, and the number of equivalent mutants is E. The mutation

score is defined to be:

$$MS(P, T) \equiv \frac{D}{(M - E)}.$$

The TRITYP program in Fig. 6 has 27 executable statements, 10 different branches, and 10 distinct feasible execution paths[4]. The Mothra system creates 970 mutants of TRITYP, 110 of which are equivalent. In an experiment reported elsewhere [13], Godzilla created 420 test cases for TRITYP that killed 862 mutants, yielding a mutation score of 0.99. These results, along with those from four other programs, are shown in Table III. The *Size* column gives the number of executable lines of code, *TCs* gives the number of test cases generated, and *M, D, E* and *MS* give the total number of mutants, dead mutants, equivalent mutants, and the mutation score. The *Time* column is the time to generate the test cases in wall-clock minutes and seconds.

Practical experience has shown that it is extremely difficult and time-consuming to manually create test data that scores above 95% on a mutation system. We spent approximately 30 h constructing 45 test cases to kill the mutants for TRITYP. Godzilla, on the other hand, created path expressions, necessity constraints and predicate constraints, solved the constraints to generate test cases, and used Mothra to execute and kill 99% of TRITYP's mutants within about half an hour of wall-clock time on a DEC MicroVax II. This was all done automatically. Given those results, only a few minutes were needed to manually find test cases to kill TRITYP's remaining eight mutants.

---

[3] This can be easily shown by induction on the path expressions used by Godzilla, which are based on predicate transformation rules for generating verification conditions [26].

[4] There are actually 121 distinct paths through TRITYP's control flow graph, but because of the structure of the program, only 10 paths are feasible.

We have repeated this experiment for dozens of unit and module-level FORTRAN subprograms of up to a few hundred lines. The test data that Godzilla generated for these programs had an average mutation score of 0.97, with very little deviation. In fact, none of the program's test data had a mutation score of less than 0.90. The wall-clock execution time for generating these test data sets (on an unloaded Sun 3/50) varied from 22 s for the smaller subroutines, to around 20 min for subprograms of 100 to 200 lines. Thus by any measure, a constraint-based test data generator can yield a significant savings in time, effort, and expense for generating quality test data for unit testing.

VI. CONCLUSIONS

Although mutation analysis has repeatedly been demonstrated to be an effective method for testing software [15], [17], one shortcoming of the technique has always been that it does not, by itself, generate test data. In this paper we have presented an automatic test data generation technique that solves this problem in a truly general way. By generating test data specifically to kill mutant programs, we get test data with the same quality as data generated interactively, but with none of the pain and expense of that interaction.

Adequacy has long been promising as a theoretical basis for testing software. The fact that our first attempt to implement a system to produce mutation-adequate test data has yielded a tool that is so successful is exciting. The development of this technology is still in its infancy, yet it already seems to provide better test data than other testing techniques. This is not because it is revolutionary, but because it is evolutionary. Constraint-based testing incorporates other testing techniques directly. Because of the mutation analysis basis of CBT, the test data generated includes the fault-detection capabilities of such testing methods as branch coverage, predicate domain analysis, and extremal values testing. The experiments that have been performed with Godzilla encourage us that CBT will create high-quality test cases that score well on the mutation system.

Using constraints to develop test data allows us to combine path-coverage techniques and symbolic execution with mutation analysis. Moreover, the way we define the necessity constraints gives us the ability to detect arbitrary kinds of faults—the necessity constraints are formed by a table-driven system that can easily be modified and extended to test for additional faults. It is our belief that constraints capture a deep and essential property of programs. As data flow analysis and symbolic execution have uses beyond test data generation, we anticipate other uses for constraints. For example, we are currently developing methods to use constraints to detect equivalent mutants and to analyze software specifications.

The theoretical aspects of software testing summarized in the introduction of this paper are both daunting and exciting. Daunting, because we know we can never demonstrate correctness of a program through software testing. Yet also exciting, because we can expect to continue to find and develop new and better techniques for testing computer programs. For example, CBT only approximates relative adequacy, still leaving some

work for the tester, so we will continue to improve the technique to try to reduce the tester's work. One extension we are currently adding is an *expression constraint*, similar to the predicate constraint, which forces entire expressions to have an incorrect value.

As stated in the introduction, our ultimate goal is to completely automate the software testing process. This work represents a way of automating the generation of test data that approximates relative adequacy—an important step toward this goal.

REFERENCES

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis," School of Inform. and Comput. Sci., Georgia Instit. Technol., Atlanta, Tech. Rep. GIT-ICS-79/08, Sept. 1979.

[2] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations," Dept. Comput. Sci., Yale Univ., New Haven, CT, Res. Rep. 276, 1979.

[3] M. S. Bazaraa and J. J. Jarvis, *Linear Programming and Network Flows.* New York: Wiley, 1977.

[4] T. A. Budd, "Mutation analysis of program test data," Ph.D. thesis, Yale Univ., New Haven CT, 1980.

[5] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, Nov. 1982.

[6] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 215–222, Sept. 1976.

[7] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A comparison of data flow path selection criteria," in *Proc. 8th Int. Conf. Software Eng.* (London UK), Aug. 1985. Los Alamitos, CA: IEEE Computer Soc., 1985, pp. 244–251.

[8] L. A. Clarke and D. J. Richardson, "The application of error-sensitive testing strategies to debugging," in *Symp. High-Level Debugging, ACM SIGSOFT/SIGPLAN*, Mar. 1983, pp. 45–52.

[9] W. M. Craft, "Detecting equivalent mutants using compiler optimization techniques," Master's thesis, Dept. Comput. Sci., Clemson Univ., Clemson SC, 1989.

[10] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, "An extended overview of the Mothra software testing environment," in *Proc. 2nd Workshop on Software Testing, Verification, and Analysis* (Banff, AB, Can.), July 1988. Los Alamitos, CA: IEEE Computer Soc., 1988, pp. 142–151.

[11] R. A. DeMillo, E. W. Krauser, R. J. Martin, A. J. Offutt, and E. H. Spafford, "The Mothra tool set," in *Proc. 22nd Hawaii Int. Conf. System Sciences* (Kailua-Kona HI), Jan. 1989, pp. 275–284.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer,"*IEEE Computer*, vol. C-11, pp. 34–41, Apr. 1978.

[13] R. A. DeMillo and A. J. Offutt, "Experimental results of automatically generated adequate test sets," in *Proc. 6th Ann. Pacific Northwest Software Quality Conf.* (Portland OR), Sept. 1988, , pp. 209–232.

[14] R. A. DeMillo, F. G. Sayward, and R. J. Lipton, "Program mutation: A new approach to program testing," in *Infotech International State of the Art Report: Program Testing.* Infotech Int., 1979, pp. 107–126.

[15] R. A. DeMillo and E. H. Spafford, "The Mothra software testing environment," in *Proc. 11th NASA Software Eng. Lab. Workshop,* Goddard Space Ctr., Dec. 1986.

[16] P. G. Frankl, "The use of data flow information for the selection and evaluation of software test data," Ph.D. thesis, Courant Instit. Mathematical Sci., New York Univ., New York, NY, 1987.

[17] M. R. Girgis and M. R. Woodward, "An experimental comparison of the error exposing ability of program testing criteria," in *Proc. Workshop on Software Test.*, July 1986. Los Alamitos, CA: IEEE Comput. Soc., 1986, pp. 64–73.

[18] M. J. Harrold, R. Gupta, and M. L. Soffa, "TBM: A testbed management tool," in *Proc. 7th Int. Conf. Testing Comput. Software, ACM SIGSOFT* (San Francisco, CA), June 1990, pp. 47–56.

[19] J. R. Horgan and A. P. Mathur, "Weak mutation is probably strong mutation," Software Eng. Res. Ctr., Purdue Univ., West Lafayette, IN, Tech. Rep. SERC-TR-83-P, Dec. 1990.

[20] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208–215, Sept. 1976.

[21] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 371–379, July 1982.

[22] W. E. Howden,*Functional Programming Testing and Analysis.* New York: McGraw-Hill, 1987.

[23] J. C. Huang, "An approach to program testing," *ACM Comput. Surveys*, vol. 7, no. 3, pp. 113–128, Sept. 1975.

[24] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[25] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing," *Software Practice and Experience*, to be published.

[26] Z. Manna, *Mathematical Theory of Computation.* New York: McGraw-Hill, 1974.

[27] L. J. Morell, "A theory of error-based testing," Ph.D. thesis, Univ. Maryland, College Park, Tech. Rep. TR-1395, 1984.

[28] L. J. Morell, "Theoretical insights into fault-based testing," in *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*(Banff, AB, Can.), July 1988. Los Alamitos, CA: IEEE Computer Soc., 1988, pp. 45–62.

[29] G. Myers, *The Art of Software Testing.* New York: Wiley, 1979.

[30] A. J. Offutt, "Automatic test data generation," Ph.D. thesis, Georgia Instit. Technol., Atlanta, 1988 (Tech. Rep. GIT-ICS 88/28) (also published as Software Eng. Res. Ctr., Purdue Univ., West Lafayette, IN, Tech. Rep. SERC-TR-25-P).

[31] A. J. Offutt, "An integrated system for automatically generating test data," in *Proc. 1990 Conf. Systems Integration* (Morristown, NJ), Apr. 1990. Los Alamitos, CA: IEEE Computer Soc., 1990, pp. 694–701.

[32] P. Purdom and C. Brown, *The Analysis of Algorithms.* New York: Holt, Reinhart and Winston, 1985.

[33] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the automated generation of program test data," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 293–300, Dec. 1976.

[34] D. J. Richardson and M. C. Thompson, "The relay model for error detection and its application," in *Proc. 2nd Workshop on Software Testing, Verification, and Analysis* (Banff, AB, Can.), July 1988. Los Alamitos, CA: IEEE Computer Soc., 1988, pp. 223–230.

[35] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 247–257, May 1980.

Richard A. DeMillo received the Ph.D. degree in information and computer science from the Georgia Institute of Technology, Atlanta.

He is a Professor of Computer Sciences and Director of the Software Engineering Research Center at Purdue University, West Lafayette, IN. He is also Director of the U.S. National Science Foundation's Computer and Computation Research Division. His research interests include the problems of software testing, safety, and reliability.

A. Jefferson Offutt received the Ph.D. degree in computer science from the Georgia Institute of Technology, Atlanta.

He is an Assistant Professor at Clemson University, Clemson, SC, where his current research interests include program testing, software reliability, module and specification testing, and software engineering environments.

Dr. Offutt is a member of the ACM and the IEEE Computer Society.