An Extended Overview of the Mothra Software Testing Environment

R. A. DeMillo, Purdue University
D. S. Guindi, W. M. McCracken, A. J. Offutt, Georgia Institute of Technology
K. N. King, Georgia State University

ABSTRACT

Mothra is a software testing environment that supports mutation-based testing of software systems. Mothra is interactive; it provides a high-bandwidth user interface to make software testing faster and less painful. Mothra currently runs on a variety of systems under 4.3BSD UNIX, UNIX System V, and ULTRIX-32 1.2. This paper begins with a brief introduction to mutation analysis. We then take the reader on a guided tour of Mothra, emphasizing how it interacts with the tester. Then we present with a short discussion of Mothra's internal design. Next, we discuss some major problems with using mutation analysis and discuss possible solutions. We conclude by presenting a solution to one of these problem—a new method of automatically generating mutation-adequate test data.

Keywords: Software tools, software testing, mutation analysis, user interface, Mothra, test data generation.

1. MUTATION ANALYSIS

Software testing attempts to provide a partial answer to the following question:

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

Several techniques have been devised for generating test cases, including input space partitioning [26], symbolic testing [30], and functional testing [27]. (See [4, 18] for a survey of these and other techniques.) Because software testing is insufficient to guarantee program correctness [26], these techniques do not attempt to establish absolute program correctness but to provide the tester with some level of confidence in the program.

Although each of these techniques is effective at detecting errors in the program, *mutation analysis* [16, 17] goes one step further by supplying the tester with information in the absence of known errors. This unique ability helps the tester predict the reliability of the program and indicates quantitatively when the testing process can end. Mutation analysis has been shown analytically and experimentally to be a generalization of other test methodologies [10, 23]. Thus, a mutation analysis testing tool gives a tester the capabilities of several other test techniques as well as features that are unique to mutation. Moreover, a new technique for creating test data allows the tester to automatically generate test data that is based on mutation analysis. The rest of this section describes the mutation approach to software testing.

1.1 Mutation Analysis in Theory

Mutation analysis is a powerful technique for software testing that assists the tester to create test data and then interacts with the tester to improve the quality of the test data. The technique is briefly described here, but the reader is referred to [10, 17] for an introductory treatment.

Mutation analysis is based on the "competent programmer hypothesis"—the assumption that the program to be tested has been written by a competent programmer. Therefore, if the program is not correct, it differs from a correct program by at most a few small errors. Mutation analysis allows the tester to determine whether a set of test data is adequate to detect these errors. The first step in mutation analysis is the construction of a collection of *mutants* of the test program. Each mutant is identical to the original program except for a single syntactic change (for example, replacing one operator by another or altering the value of a constant). Such a change is called a *mutation*. Each mutant is then executed, using the same set of test data each time. Many of the mutants will produce different output than the original program. The test data is said to *kill* these mutants; the data was adequate to find the errors that these mutants represent. Some mutants, however, may produce the same output as the original program. These *live* mutants provide valuable information. A mutant may remain alive for one of two reasons:

- *The test data is inadequate.* The test data failed to distinguish the mutant from the original program. For example, the test data may not exercise the portion of the program that was mutated.
- *The mutant is equivalent to the original program.* The mutant and the original program always produce the same output, hence no test case can distinguish between the two.

Normally, only a small percentage of mutants are equivalent to the original program. More test cases can be added in an effort to kill nonequivalent mutants. The adequacy of a set of test cases is measured by an *adequacy* score; a score of 100% indicates that the test cases kill all nonequivalent mutants.

Although mutation analysis tests only for simple errors, theoretical studies have shown that simple and complex errors are coupled (this is called the coupling effect), hence test data that causes simple nonequivalent mutants to die will usually also cause complex mutants to die [2, 17].

We will use the Ada function in Figure 1 to illustrate these concepts.

```
function MAX(M, N: INTEGER)
  return INTEGER is
begin
  if M > N then
    return M;
  else
    return N;
  end if;
end MAX;
```

Figure 1. Function MAX

One way to mutate this function is to replace the > operator in the **if** statement by some other relational operator (>=, <, <=, =, or /=). This produces a collection of five mutants of MAX, which we can attempt to kill by supplying appropriate test data. As a first try, we might choose the values M = 1, N = 2. The original function returns 2; the mutants return the following values:

if M >= N **then**: 2
if M < N **then**: 1
if M <= N **then**: 1
if M = N **then**: 2
if M /= N **then**: 1

Three mutants return different values and are therefore considered dead. The >= and = mutants are still alive. The former mutant remains alive because it is equivalent to the original function; no set of test data can kill this mutant. The latter mutant can be killed by a second set of test data, such as M = 2, N = 1.

1.2 Mutation Analysis in Practice

A mutation-based testing system allows a tester to perform mutation analysis on a program (or subprogram(s)). The tester supplies the program to be tested (the program is assumed to be valid according to the rules of the source language) and chooses the types of mutations to be performed. The tester also supplies one or more test cases. The testing system executes the original program and each mutant on each test case and compares the output produced by the two programs. If the output of a mutant differs from the output of the original program, the mutant is marked dead. Once execution is complete, the tester can examine any mutants that are still alive. The tester can then declare a mutant to be equivalent to the original program, or the tester can supply additional test cases in an effort to kill the mutant (and possibly other live mutants as well). Some mutation systems are able to detect automatically certain kinds of equivalent mutants.

Several mutation systems have been built over the past ten years, including PIMS [10], EXPER [2, 8, 11], and FMS.3 [36], which supported mutation analysis for Fortran programs, and CMS.1 [1, 2, 24], which handled Cobol. The most recent mutation-based system is Mothra [19], an integrated software testing environment under joint development at Purdue University and Georgia Tech's Software Engineering Research Center. Mothra has several notable features that previous mutation systems lack:

- *Mothra provides a high-bandwidth user interface.* Testing requires a great deal of interaction between the tester and the testing system. Mothra's interface makes testing less of a chore while allowing the presentation of more information on the display, thereby encouraging the tester to do a better job.
- *The architecture of Mothra imposes no* a priori *constraint on the size of software systems that can be tested.* Mothra supports the testing of programs ranging in size from 10 lines to 100,000,000 lines. The maximum program size is limited only by the speed and storage capacity of the machine on which the testing is being done and on the parser used.
- *Mothra is designed to support multiple source languages.* At present, Mothra supports the testing of Fortran 77 programs; work is underway to extend Mothra to allow the testing of programs written in other languages, including Ada.
- *Mothra allows easy incorporation of new tools.* Currently, these include a test case generator (to help the tester create test cases to kill particular mutants, described in Section 4) an automatic equivalence checker (to detect equivalent mutants) and an oracle.

Experience with Mothra and earlier systems has shown mutation analysis to be a powerful tool for program testing. Mutation-based testing systems have a number of attractive features:

- *Mutation analysis includes—as special cases—most other test methodologies.* Statement coverage and branch coverage are among the methodologies that mutation analysis subsumes.
- *A mutation-based system provides an interactive test environment that allows the tester to locate and remove errors.*
- *Mutation analysis allows a greater degree of automation than most other testing methodologies.*
- *Mutation analysis provides information that other test methodologies do not.* In particular, the mutation score for a particular program indicates the adequacy of the data used to test the program, thereby serving as a quantitative measure of how well the program has been tested.

A potential problem of mutation-based testing systems is the amount of computer resources (both space and time) required for the testing of large programs. The number of mutants generated for a program tends to grow quadratically with the number of names in the program. Storing huge numbers (perhaps millions) of mutants can be difficult on many computer systems; executing that many mutants is an even larger problem. Fortunately, there are several ways to overcome the problem of limited resources:

- *Mutant sampling.* Mutation systems allow the tester to randomly sample a certain percentage of mutants. Often, sampling even a small percentage of the possible mutants is enough to construct very effective test data.
- *Selective application of mutant operators.* Mutation systems also allow the tester to specify that only certain kinds of mutations are to be performed. Thus, the tester can select mutations that are likely to have a high payoff relative to the amount of time they require. This feature also allows the system to "mimic" the testing capabilities of other testing techniques. It also provides the test the testing manager with an effective way of distributing the testing effort.
- *Use of high-performance computers.* The Mothra system is designed to support *resource shifting*—the automatic transfer of computationally intensive tasks (such as mutant execution) to high-speed processors (vector machines, for example).

We feel that mutation analysis is particularly useful for testing mission-critical applications, where the testing of programs must be extremely thorough. Mutation analysis is not only a superior testing methodology, but also offers testers the opportunity to match the degree of testing to the criticality of the application and the amount of resources available during testing.

2. INTERACTING WITH MOTHRA

In this section, we describe the capabilities of the Mothra system by showing how a tester would interact with it. (A more complete discussion of how to use Mothra appears in [6].) As a running example, we will use Mothra to test a binary search routine from Kernighan and Plauger's *The Elements of Programming Style* [29, p. 107]. In [29], the routine appears as a fragment of PL/I code; we have converted the fragment to a Fortran 77 function named BSEARCH in Figure 2.

We chose this example because it is small, familiar, and contains a single (intentional) bug, making it a good choice for an introduction to Mothra. (The bug occurs in the comparison of HIGH with LOW: .LE. should be .LT.)

Mothra has several different interfaces; this paper describes an interface that runs on top of the X Window System [22]. The high bandwidth of X enables Mothra to supply the tester with a great deal of useful information. Figure 6 (printed after the reference page) shows a snapshot of the Mothra display under X. As we examine each step in the testing process, we will refer to relevant windows in Figure 6.

```
      LOGICAL FUNCTION BSEARCH(TABLE,LIMIT,SEARCHA)

      INTEGER LIMIT, TABLE(LIMIT), SEARCHA, LOW, HIGH, MID

      LOW = 1
      HIGH = LIMIT
10    MID = (LOW + HIGH) / 2
      IF (HIGH .LE. LOW) THEN
        BSEARCH = .FALSE.
        RETURN
      ELSE
        IF (SEARCHA .EQ. TABLE(MID)) THEN
          BSEARCH = .TRUE.
          RETURN
        ELSE
          IF (SEARCHA .GT. TABLE(MID)) THEN
            LOW = MID + 1
          ELSE
            HIGH = MID - 1
          ENDIF
          GOTO 10
        ENDIF
      ENDIF

      END
```

Figure 2. Function BSEARCH

2.1 Starting an Experiment

In Mothra, an *experiment* is the process of generating a set of test data for a given program or subprogram (in our case, the BSEARCH function). A Mothra experiment involves three steps:

- *Test preparation.* The tester selects mutant operators and enters test cases.
- *Mutant execution.* Mothra executes the mutants on the test cases supplied by the tester. Mothra compares the output of each mutant with the output produced by the original program (on the same test case), marking the mutant dead if the output differs.
- *Analysis.* The tester analyzes the results of mutant execution and decides whether or not the test data generated meets the criteria for ending the experiment. (In our discussion, we assume that the goal is to kill 100% of all nonequivalent mutants. In practice, the stopping criteria will often be less ambitious; for example, the goal might be to kill a certain percentage of mutants or to kill all mutants of selected types.)

These steps must occur sequentially, but they can be repeated any number of times. If, during analysis, the tester decides that the test data is not yet good enough, the process can be repeated, starting with test preparation, then mutant execution, and so forth. Thus, Mothra guides the tester to produce adequate test data by interactively measuring the test data and, through the live mutants, pointing out specific ways in which the data is inadequate.

At the beginning of an experiment (before test preparation), the tester must choose a name for the experiment. From the experiment name, Mothra creates names for the data various files that it opens during the course of the experiment. The experiment name is also useful when an experiment is to be suspended temporarily and later resumed. The tester must also give Mothra the name of the file containing the program to be tested. (Note: Mothra can be used to test either complete programs or individual subprograms. For simplicity, we will always use the term "program" to refer to the piece of software undergoing testing.) Mothra immediately translates the program to an internal form (described in [33]) that is used throughout the experiment.

2.2 Test Preparation

Test preparation involves two separate actions: enabling mutants and entering test cases. The tester may perform these steps in either order, but must do both before mutant execution can begin.

2.2.1 Enabling Mutants

Mothra mutates a program by applying a *mutant operator*—a simple transformation—to the program. Each mutant operator generates a number of mutant programs, each one identical to the original program except for a minor change to one line of the program. Mothra supplies twenty-two types of mutant operators for Fortran 77. We have grouped these operators into eight *classes*, which are further grouped into three *levels*. The following is an abbreviated list of Mothra mutant operators, arranged by level.

- *Statement analysis*: replace each statement by TRAP (executing the TRAP instruction will kill the mutant); replace each statement by CONTINUE; replace each statement in a subprogram by RETURN; replace the target label in each GOTO; replace the label in each DO statement.
- *Predicate and domain analysis*: take the absolute value or negative absolute value of an expression; replace one arithmetic operator by another; replace one relational operator by another; replace one logical operator by another; insert a unary operator preceding an expression; alter the value of a constant; alter a DATA statement.
- *Coincidental correctness*: replace a scalar variable, array reference, or constant by another scalar variable, array reference, or constant; replace a reference to an array name by the name of another array.

The hierarchical classification of mutant operators allows a functional partition of the experiment when multiple testers are involved. The testing of a piece of software can easily be distributed over several testers by making each tester responsible for using a specified set of mutant levels or classes, for example.

The classification scheme also provides a logical partition of the experiment. Because it is based on mutation analysis, Mothra supports several different kinds of testing. Each class, level, or individual mutant operator corresponds to a particular kind of test. For example, by constructing test cases that kill mutants generated by the statement analysis level, the tester performs traditional statement analysis tests (testing that all statements are executed and that every statement has an effect, for example). The predicate and domain analysis level can be used to create test data to exercise all predicate boundaries and data domains. Using the coincidental correctness analysis level provides test data that eliminates coincidental correctness errors (small errors in the original program that have gone undetected because, by the nature of the test data, the program has coincidentally produced correct results).

Prior to mutant execution, the tester must select a collection of mutant operators to be applied to the original program. The tester can select all mutant operators, all operators in a particular level, all operators in a class, individual operators, or any combination of these. In the X interface, the tester selects a level by pointing to an icon representing that level and clicking (see the lower left window in Figure 6). To select all mutant operators, the tester selects all three levels. To select an entire class or any subset of the operators in a class, the tester points to the class icon and clicks. Icons representing the operators in the class then appear on the following line. The tester can use the mouse to select a subset of these; a special ALL icon allows the tester to select all operators in a class.

When the tester has finished selecting mutant operators, Mothra generates a file of *mutant descriptor records*, which describe how the internal form of the original program is to be modified to create the desired mutants. We describe this process in Section 3.

When selecting mutant operators for first time, the tester must also choose a *test strength*. The test strength is a number between 0 and 100 representing the percentage of generated mutants that the tester wants to *enable*. (Enabled mutants are executed during the mutant execution step; mutants not enabled are ignored.) The default test strength is 100 (all generated mutants are enabled). Test strengths of less than 100 are useful when testing very large programs. The mutants to be

enabled are then chosen randomly from the set of generated mutants. However, if two experiments use the same original program, the same mutant operators, and the same test strength, then the same mutants will be enabled. This ensures that Mothra experiments will be repeatable.

When we tested BSEARCH, we enabled all mutants at a test strength of 100% (see the lower left window in Figure 6). After generating the mutant descriptor records, Mothra showed us that 307 mutants were generated (see the top line of the window at upper right).

2.2.2 Entering Test Cases

The other task to be performed during test preparation is entering test cases. A test case may include two kinds of values: *initial values* and *run-time values*. When testing a subprogram, the tester must supply initial values for its arguments. When testing a complete program, the tester can supply initial values for any variables that the program does not initialize. Run-time values are values that the program requests during execution by means of READ statements; the tester can either enter run-time values manually or have Mothra read them from a file.

After the tester has entered the initial values belonging to a test case, Mothra executes the original program with the test case as input. When the program executes a READ statement, Mothra waits for the tester to enter the data requested by the statement, or optionally reads the data from the supplied input file. When the program has completed execution, Mothra displays the program's output (called the *expected output*) and asks the tester if it is correct. If the expected output is not correct, there is an error in the original program and testing should be suspended until the error has been located and fixed. Once the problem is fixed, the testing would have to commence again. If the output is correct, Mothra saves it for use during the mutant execution phase.

The first six test cases used in our BSEARCH experiment appear in the upper middle window of Figure 6. Mothra has truncated "SEARCHA" to "SEARCH" because of Fortran 77's requirement that identifiers not be longer than six characters.

We have also implemented a test case generator that can automatically produce test cases for Mothra. This tool removes the tester from one of the most time consuming and demanding facets of using Mothra. This generator is described in Section 4.

2.3 Mutant Execution

Mutant execution begins when the tester has supplied Mothra with a program to be tested, a set of enabled mutants to be executed, a test strength, a set of test cases, and a signal to start mutant execution. Mutant execution is the only one of the three phases that is completely automated by the Mothra system. It requires no interaction from the tester once execution has begun.

During mutant execution, Mothra creates each mutant program, executes it on a test case, and compares its output with the expected output for that test case. If the results differ, the mutant is marked dead. Dead mutants are not executed against subsequent test cases. If the mutant remains alive, it will be executed against all test cases supplied until it dies or all test cases have been tried.

As execution progresses, Mothra displays a collection of statistics describing the number of mutants that were killed compared with the total number of enabled mutants. These statistics are the main indicators of the effectiveness of the test cases. The tester can either get a high-level view of the experiment status (the overall mutation score, as well as the score for individual mutation classes) or a more detailed set of statistics for each mutant operator. Regardless of which view the tester chooses, Mothra continuously updates the percentage of mutants executed. In Figure 6, the window at upper right shows the high-level view of our example experiment.

2.4 Analysis

The objective of analysis is to determine whether or not enough mutants have been killed to satisfy the tester's stopping criteria. During analysis, the tester first studies the statistics provided by Mothra. If the percentage of mutants killed is sufficient, the tester may not need to proceed further. More often, however, the percentage will be too low, and the tester will be required to add additional test cases in an attempt to kill more mutants. Alternatively, the tester may choose to examine the remaining live mutants in an attempt to determine whether any are equivalent to the original program. (At present, Mothra provides no automatic method of detecting equivalent mutants.)

Equivalencing mutants is particularly easy with the X interface. At the tester's request, Mothra displays a copy of the original program, with live mutants shown as single lines of code marked with the symbol # (see the upper left window in Figure 6). The tester can then equivalence a block of one or more mutants by using the mouse to select the mutants, then pointing to the "Equivalence" icon at the bottom of the window. A scroll bar at the left edge of the window allows the tester to move through the program quickly.

Most equivalent mutants are easy to spot. For example, if I is a program variable that never takes on negative values, then any mutant in which I is replaced by ABS(I) is equivalent to the original program. Detecting some equivalent mutants requires a more detailed understanding of the program, however. In BSEARCH, for example, one mutant is created by changing the statement

```
MID = (LOW + HIGH) / 2
```

to be

```
MID = (LOW + LOW) / 2
```

Normally we would not expect this change to produce an equivalent program. In BSEARCH, however, it does, because of the binary search algorithm that BSEARCH uses. Setting MID (the midpoint of the search) to LOW (as the latter statement effectively does) may reduce the speed of the search, but does not cause BSEARCH to fail.

Equivalent mutants are, in some ways, treated like dead mutants. Once the tester has declared a mutant equivalent, Mothra does not execute the mutant against test cases during mutant execution, nor does Mothra include the mutant in the live mutant count. Mutants should be equivalenced as soon as possible, because equivalencing a mutant prevents the re-execution of the mutant each time the tester adds a new test case.

After analysis, the tester may choose to terminate the experiment (if the mutation scores indicate that the test cases are of sufficiently high quality) or continue with a new test preparation step. During this step, the tester may decide to enable more mutants (unless all mutant operators have already been selected). Whenever the tester enables new mutants, they will be executed against all existing test cases plus any new test cases that the tester has added.

In our BSEARCH experiment, we proceeded one test case at a time. The first test case killed 172 mutants, leaving 135 still alive. After the second test case, 114 mutants remained alive. We continued to add test cases until the number of live mutants became manageable. We then asked Mothra to display the remaining mutants so that we could locate equivalent mutants; we found 34. Next, we continued adding test cases to try to kill the remaining live mutants. After a total of eleven test cases, only two mutants remained (see the upper left window in Figure 6). In an attempt to kill these mutants (which, incidentally, are equivalent to each other), we tried the following test case, which causes LOW and HIGH to have the same value.

```
Values for case 12.
TABLE  1
LIMIT  1
SEARCH 1
```

This test case causes BSEARCH to give the wrong answer, so we found the error in the function. The final two mutants are actually the correct version of BSEARCH.

A few final remarks about the BSEARCH example are in order. First, notice that the number of equivalent mutants (34) is approximately 10% of the total number. This percentage is typical of most programs, although it may vary somewhat depending on the mutant operators selected, the programming style present in the original program, the algorithm used, and other factors. Second, we must confess that, for the sake of this example, we deliberately avoided test cases that would have exposed the error in BSEARCH sooner. Using Mothra, an experienced tester should find the error within the first few test cases. Third, if we redo the BSEARCH experiment with the error corrected, we find that the original twelve test cases do not kill all of the nonequivalent mutants; we must add a few additional test cases. In general, any change to a program—no matter how small—affects the test cases needed to kill the program's mutants.

3. PROBLEMS WITH MUTATION ANALYSIS

Although mutation analysis has been shown experimentally [23] and analytically [8] to be a very effective means of testing software, it has historically suffered from shortcomings. Part of the goals of the Mothra project is to provide solutions to these problems. In this section, we discuss three of the problems with practically applying mutation analysis and indicate the research that is being carried out to solve these problems. In the next section, we present some results from one of these efforts; specifically, a means of generating test data to score well in a mutation system.

3.1 Resource Consumption

One problem with mutation analysis that has been mentioned is that of the computer resources that are consumed by mutation. Mutation analysis systems must create and execute many mutations of the program under test. For example, the function BSEARCH shown in Figure 2 is a 20 line program that generates 307 mutants with all mutant types enabled under the Mothra system. In general, the number of mutations of a program is bounded by the square of the number of references to variables and constants [8].

Remember that mutation systems execute each (live) mutant against each test case. Executing this many mutant programs can be a quite expensive and time consuming process. Several suggestions have been made to limit this execution cost. Howden first suggested the concept of *weak mutation* [28] as a way of applying mutation by executing components of programs rather than complete programs. An obvious advantage of weak mutation over strong mutation is that much of the execution time can be saved. A disadvantage of weak mutation is that complete weak mutation test sets are effective only for components of the program rather than the entire program.

Another means that has been suggested of shortening the time spent executing mutants of a program is that of *combined stream execution* [33]. During mutation analysis, the test program is executed many times—once in the original form and once for each mutant. Since each mutant changes only one statement in the program, the execution stream of a mutant program is identical to that of the original version up to the point where the mutated statement is executed. A combined-stream execution approach takes advantage of the resultant duplication of effort by splitting the execution stream of the original program to begin mutant execution at the point where the mutated statement appears in the program.

Current research is also being undertaken to attempt to parallelize mutation [31, 32]. This involves development of parallel algorithms to execute mutant programs.

3.2 Oracle

A second problem with mutation analysis (and indeed with most software testing techniques) is that of determining whether the output of a program on a specified data point is correct. The *oracle* is a term that is applied to some entity that can recognize whether output is correct. The oracle function can be automated by a set of specifications, sample data, or another program. In practice, and in current mutation systems, the oracle is usually the tester. We are currently looking at ways to automate this function within Mothra by using such things as predicate functions to decide whether the output of a program is correct.

3.3 Test Data Generation

Most software testing techniques generate test data that satisfies some criteria (such as executing all branches). Mutation analysis is a little different in that it measures test data rather than creates test data. The test data is assumed to come from an external source—in previous mutation systems that external source has been the tester. The mutation systems were used to interactively guide the tester to create a "high quality" set of test cases, where "high quality" test cases detect and kill mutations. Unfortunately, the actual creation of test cases has been the most human-intensive and therefore the most expensive action within mutation systems. In the next section, we describe in some detail a method for automatically generating test data for Mothra.

4. GENERATING MUTATION-ADEQUATE TEST DATA AUTOMATICALLY

Generating test data, whether within a mutation system or without, is a difficult task posing complex problems. In fact, the general problem is unsolvable [9]. Thus, practical test data generation techniques tend to choose a subset of the possible inputs according to some rationale. The assumption is that the subset of inputs chosen has the properties of being able to find a large portion of the errors in the program as well as being able to establish some confidence in the software. A set of test cases that score well on a mutation system will have just those properties—the test data will find the set of errors represented by the mutant operators and, if executed successfully, the software is ensured to have none of those errors.

We hereby describe a methodology which uses the concepts of mutation analysis to automatically generate test data. This test data is designed specifically to kill the set of mutations that are defined on the program. Such test data can be used to kill mutants within a mutation system or used independently of mutation as a set of effective test cases. Moreover, because of the mutation analysis basis, such test data will be assured of having such properties as containing extremal values, covering all statements and all branches. As a matter of fact, studies [23, 37] show that with proper modification, data that score well on a mutation system can be expected to be more general than any other test data generation method.

The concept of using mutants to generate specific tests was also suggested by Budd [8], by Howden [28] in his "weak mutation" approach, and by Foster [20] in his error sensitive test case analysis. The present work, however, represents what we think is the first general and implemented attempt to generate adequate test sets.

In generating test cases to kill a specific mutant, the set of possible test cases can be divided into two subsets. One kind of test case for a mutant will differentiate the mutant from the original program (killing the mutant). The other kind of test case will not. The process of creating a test case to kill a mutant can be described as selecting the first kind of test case from the set of possible test cases. To automatically create such test data, we need some kind of filter that filters out ineffective test cases, leaving the effective ones. **This filter can be described by a set of *constraints* on a test case, where an effective test case must satisfy those constraints.**

4.1 Killing Mutants

This section describes the characteristics of an effective test case filter. The program MAX is used throughout this section as an example of the concepts. MAX is repeated in Figure 3 for convenience. It

is shown with one embedded mutant that will also be used for the examples.

```
function MAX(M, N: INTEGER)
  return INTEGER is
begin
  if M > N then
    return M;
#   return N;
  else
    return N;
  end if;
end MAX;
```

Figure 3. Function MAX

To discover how to select a test case that kills an individual mutant, we must first realize that the mutant is represented as a syntactic change to a particular statement. Since every other statement in the mutated program is exactly the same as in the original program, it is apparent that as a minimum we must execute the mutated statement. Assuming the mutated statement is reached, the test case must be such that the state of the original program after executing the mutated statement must differ from the state of the mutant program after 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 the end of the program.

To simplify this discussion, for a test case T to kill a mutant M appearing on line S of a program P, T has to have three broad characteristics. Note that these characteristics make increasingly specific requirements on the test case:

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

Each of these three characteristics are discussed below.

4.1.1 Reachability

To generate test cases that are guaranteed to reach a particular statement S, we need some expression that describes a path to be taken through the program. This is similar to statement analysis and can be found discussed elsewhere (see, e.g., [12, 26]).

4.1.2 Necessary

The *state* of a program is described by the value of program variables and internal variables such as the program counter. In order to kill a mutant, a test case must create a state in the mutant program that differs from the state of the original program. Since the syntax of the mutant program and the original program are identical except for the mutated statement, if the states of the two versions of the program are equal after execution of the mutated statement, then the mutant will not die. Note that the necessary condition is reminiscent of Howden's weak mutation [28].

In the above example, the necessary condition means that the return value of the function MAX in the mutant program must differ from the return value of MAX in the original program. Since the return value is the value of M in the original program and N in the mutant program, the necessary condition for this mutant would require that M and N have different values.

The necessary state difference can be described in terms of the same variables and program symbols that the mutant effects. Thus, this change depends solely on the mutant operators for the language. A set of conditions, called *necessary constraints*, has been defined that a test case must meet to kill mutants. We discuss these constraints in more detail below.

4.1.3 Sufficiency

Although a local state change is necessary to kill a mutant, it will not guarantee that the final state of the mutant program will differ from that of the original program. Once the states of the mutant program and the original program diverge, they may well converge to the same, final state. Thus, the constraint is only sufficient to kill a mutant if it ensures that the final state of the mutant will differ from that of the original program. Completely solving the sufficiency condition implies knowing in advance the path that a program will take. Although this is in general intractable, Richardson [35] is working on this problem using path analysis techniques. The approach taken here is to define a series of increasingly complex approximations to sufficient constraints.

The first approximation is to assume that if a test case meets the necessary condition, it will usually meet the sufficiency condition. This assumption is not as simplistic as it might first appear. For the necessary condition to have been met but not the sufficiency condition then the mutant did not die and the output of M was identical to that of P. Since the necessary condition was met, then the state of the mutant program M did diverge from the original program P at the point following the mutated statement. That is, the state of M first diverged from P, then returned to that of P.

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

4.2 Some Simple Constraints

The most novel aspect of this technique is the rationale that we use for constructing the test data. Because the data is designed to kill specific mutants, it will be assured of finding those types of errors covered by mutation. The actual method for constructing the data attempts to mimic the way testers construct data to kill mutants when using a mutation system interactively. The basic approach can be stated very simply: **for a test case T to kill a mutant, T must make a difference in the mutant's behavior.** That is, since the mutant is represented by a single change to the source program, the state of the mutant program must diverge from that of the original program at the point of the mutated statement.

We intend to describe a test case in terms of arithmetic expressions, and other primitive formulae that when satisfied can be used to generate values. We formalize these descriptions as constraints on the test case. These constraints are taken directly from the mutant operators defined on a language [15].

These mutant operators are described in terms of source variables, keywords, and operators. Since the operators are applied to a program as syntactic changes, a test case that meets the necessary condition for a mutant must ensure that that syntactic change results in a state change for the program.

For example, one mutant operator is the Scalar Variable Replacement operator (*svr*), which replaces scalar variable references by references to other scalar variables. In the MAX example, the first mutant shown was an *svr* mutant where M was replaced by N. A test case to kill that mutant must at least have the property that the value of the

replacement variable does not equal the value of the original variable, thus we get the constraint $M \neq N$. This constraint can be generalized to the form $X \neq Y$, where X and Y represent arbitrary scalar variables.

Thus, a set of constraint *templates* can be defined that correspond to the mutant types. The templates that are currently defined for the mutant operators defined in the Fortran 77 version of Mothra are presented in Table 1. In the table, the symbols X and Y are used for scalar variables, A and B are used for array names e_1 and e_2 are used for arbitrary expressions, ρ and ϕ are used to represent binary operators, and C is used to represent constants.

Most of these mutant types are replacement type mutants, where one program symbol is replaced by another program symbol. The constraint template requires that the value of the two program symbols differ. An exception is the *abs* mutant type. The *abs* mutant as applied to an expression actually involves three separate mutations: insertion of an absolute value operator; insertion of a NEGABS operator, which takes the negative of the absolute value; and the insertion of the ZPUSH operator. The ZPUSH operator causes an error condition to be raised if the expression is zero, thus killing the mutant. In order to kill these mutants we need the expression to take on a negative value, a positive value, and the value zero.

The operator replacement mutants (*aor, lcr, ror*) are also worth mentioning. In order to kill these mutants, the value of the expression must be different from that of the original program's expression.

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[2]	DO statement end replacement	$e_2 - e_1 \geq 2$
		$e_2 < e_1$
lcr	logical connector replacement	$e_1 \rho e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	$e_1 \rho e_2 \neq e_1 \phi e_2$
sar	scalar for array replacement	$A(e_1) \neq X$
scr	scalar for constant replacement	$C \neq X$
svr	scalar variable replacement	$X \neq Y$

TABLE 1. Constraint Templates

Several mutant types not represented in the above table are shown in Table 2. These mutant type represent changes that will by definition always satisfy the necessary constraint. For example, the *crp* mutant replaces one constant with another constant, and the obvious necessary constraint, that the two constants have different values, is independent of the test case. In fact, in the Mothra system, these mutants are not generated unless the values do differ. Likewise, *dsa* mutants, *src* mutants, and *uoi* mutants all change the value of constants and the necessary constraint is automatically satisfied. The other mutant types, *glr, rsr, san,* and *sdl* mutants change the state of the mutant program by changing the flow of control. Thus, they too automatically satisfy the necessary constraint. The only requirement of a test case to kill mutants of this type is that the test case causes the statement to be executed. That is, the test case should satisfy reachability.

2. A DO statement in Fortran 77 takes the form "DO L x = e_1,e_2" and the *der* mutant replaces the label L with other labels. Thus the constraint is $e_2 - e_1 \geq 2$, requiring that the loop be executed at least twice.

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

TABLE 2. Mutant types with no templates

We mentioned previously that the necessary constraints are reminiscent of the weak mutation analysis suggested by Howden [28] in the sense that they force a local state difference between the original and the mutant program. The major differences are that the test data is automatically created and the method described here uses the strong mutant operators rather than those suggested by Howden. He described five kinds of mutations to a program; variable reference, variable assignment, arithmetic expression, relational expressions, and Boolean expression. Although the strong mutant operators are divided differently, we can relate variable reference, variable assignment, and arithmetic expression mutants to *aar, asr, car, cnr, csr, sar, svr,* and *uoi* operators. Howden's arithmetic relation operator is exactly analogous to the *ror* mutant and his Boolean expression operator is analogous to the *lcr* mutant. In addition, strong mutation operators include the statement mutations, mutations of constants, arithmetic operators, and DO statement alterations.

4.2.1 Predicate Constraints

Early experimentation with a prototype implementation of this testing technique revealed a major inadequacy with the necessary constraints as defined. The problem was with killing mutations inside predicate expressions. Although the test cases being generated would cause an immediate effect on the state of the mutant program, the (binary) result of the mutant predicate would be the same as of the original version. An example of this problem is shown in Figure 4.

Mutant
IF (I+K.GE.J) THEN
# IF (3+K.GE.J) THEN
Necessary constraint
$I \neq 3$
Test case
I = 7, J = 9, K = 7

Figure 4. Predicate Problem

Figure 4 shows a constant for variable replacement where the constant "3" is used instead of the variable "I". Although the test case satisfies the necessary constraint (as shown), the result of the predicate test remains unchanged. $I+K=14$ and $3+K=10$, both of which are greater than J.

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

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

```
      SUBROUTINE TRITYP (I,J,K, CODE)

      INTEGER I,J,K, CODE
      INTEGER MATCH
C     CODE IS OUTPUT FROM THE ROUTINE:
C        CODE = 1 IF TRIANGLE IS EQUILATERAL
C        CODE = 2 IF TRIANGLE IS ISOSCELES
C        CODE = 3 IF TRIANGLE IS SCALENE
C        CODE = 4 IF NOT A TRIANGLE
C
C     Count matching sides
      MATCH = 0
      IF (I.EQ.J) MATCH = MATCH + 100
      IF (I.EQ.K) MATCH = MATCH + 200
      IF (J.EQ.K) MATCH = MATCH + 300
C     Select Possible scalene triangles
      IF (MATCH.EQ.0) GOTO 10
C     Select Possible isosceles triangles
      IF (MATCH.EQ.100) GOTO 20
      IF (MATCH.EQ.200) GOTO 30
      IF (MATCH.EQ.300) GOTO 40
C     Triangle must be equilateral.
      CODE = 1
      RETURN
C     Possible scalene
10    IF ((I+J).LE.K.OR.(J+K).LE.I.OR.(I+K).LE.J) GOTO 50
      CODE = 3
      RETURN
20    IF ((I+J).LE.K) GOTO 50
      GOTO 60
30    IF ((I+K).LE.J) GOTO 50
      GOTO 60
40    IF ((J+K).LE.I) GOTO 50
      GOTO 60
C     No triangle possible.
50    CODE = 4
      RETURN
C     Isosceles
60    CODE = 2
      RETURN
      END
```

Figure 5. Subroutine TRITYP

Note that this constraint extension is very similar to the notion of "firm" mutation testing [38]. In brief terms, firm mutation is a compromise between weak and strong mutation. Rather than measuring the state of a mutant program just after execution of the mutant or after execution halts, in firm mutation we measure the state of the mutant program at some point in between. The predicate constraint ensures that the state of the mutant program will differ from that of the original program at some point in between the mutant and the final state.

4.3 The Prototype Test Data Generation System

A system that implements these ideas to generate test data has been implemented at Georgia Tech as part of the Mothra testing system. Although the technique is certainly language independent, the current tool generates data for Fortran 77 programs. This system uses the rules described above to create a set of constraints for a program. A symbolic analyzer is used to create a set of path expressions. The constraints and path expressions are then conjuncted and a test case is generated to solve each conjunction. This last step is called constraint satisfaction.

A program that has been widely studied in the literature [13, 16, 7, 34] is the triangle classification program TRITYP. The Fortran program in Figure 3 takes three integers as input, representing the relative lengths of the sides of a triangle. The result is either 1, 2, 3, or 4, indicating that the input triangle is equilateral, isosceles, scalene or illegal, respectively. In [14] test data derived from five different test data generation techniques were compared in terms of the mutation score they achieved on the TRITYP program.

The results of the repeated experiment are given in Table 3. The data for the five techniques were generated by hand. The table gives the number of test cases generated for each technique, the number of mutants killed, and the mutation score for those test cases. For the statement analysis technique, for example, five test cases were generated for TRITYP. When executed against the 1031 mutant programs of TRITYP, 642 mutants produced incorrect output—killing those mutants. Since there were 69 equivalent mutants, this gave a mutation score of

$$\frac{642}{(1031 - 69)} = .667.$$

Using the Mothra mutation system, this experiment was repeated using this same version of the TRITYP program. As in the original study, only non-negative integers were generated because the program was not designed to work on negative inputs. Data was generated for TRITYP using the prototype. The data for this technique is also shown in Table 3.

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

TABLE 3. Summary of comparison results

Thus, the current prototype version of this test data generation system is producing results that are better than the five techniques studied in this previous experiment. Of course, one could argue that in comparison with domain analysis, we added 461 test cases to kill 16 mutants, surely an expensive approach. However, an examination of the 536 constraint based test cases shows that only 46 were actually effective at killing mutants, compared with 41 effective test cases from the domain testing technique. Furthermore, since the constraint based test set was generated by entirely automated methods, the number of test cases should be balanced against the labor-intensiveness of the other techniques.

As a labor comparison, the test data for the first five testing techniques were generated by hand. It required approximately 7 man-weeks to hand-generate all the data and execute the data against the mutants. The automated method took only a few hours. Moreover, this few hours was almost entirely computer time—the tester simply pushed the "go" button.

Table 4 shows the results of the test data generated for four other programs. BUBBLE is an eleven line subroutine that sorts an array of integers. CALENDAR is a 29 line subroutine that computes the number of days between two given dates that was first studied by Geller [21]. FIND is a 28 line subroutine that was first studied by Hoare [25] and later by Demillo et al. [16]. FIND takes an array of integers a input and as output produces an array such all elements to the left of a given element are less than or equal to that element and all elements to the left of a given element are greater than or equal to that element. GCD was presented by Bradley [5] and uses 55 lines to compute the greatest common divisor of an array.

Program	Number of Test Cases	Mutants Killed	Mutation Score
BUBBLE	32	304	1.00
CALENDAR	419	2624	.95
FIND	58	953	.99
GCD	325	4747	.99

TABLE 4. Results on Other Programs

5. CONCLUSIONS

We have presented two major contributions in this paper. The first is a description of the Mothra mutation system. We have explained the concept of mutation analysis and shown how it has been implemented in a practical system. We believe that mutation-based software testing environments such as Mothra will be important tools for the tester.

The second major contribution is an error based testing technique that can generate assuredly effective test data. It is very much a human based system because the constraints were derived to mimic the way people constructed test data when using a mutation system. It has the advantage, moreover, of being completely automatable.

The technique is heavily based on mutation analysis. Although mutation analysis has been repeatedly demonstrated to be an effective method for testing software, a shortcoming has always been that it does not generate test data. 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. Of course, without a solution to the problem of sufficiency, the test data generated will not be fully adequate. This problem is certainly not unique to this method or mutation analysis. As a matter of fact, Richardson and Thompson [35] are working on just this problem; there is no reason why a solution could not be integrated with a tool that generates test data using constraints.

Because this technique is based on a set of rules, these rules can be easily modified to reflect new research in software testing, knowledge about the relevance of the rules, new languages, or a host of other innovations. In particular, the capabilities of other testing techniques can be incorporated simply by expanding the set of necessary constraint templates.

References

1. Acree, A. T., *On Mutation*, Ph.D. thesis, School of Information and Computer Science, Georgia Institute of Technology, 1980.

2. Acree, A. T., T. A. Budd, R. A. DeMillo, R. J. Lipton and Sayward, F. G., *Mutation Analysis*, Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, 1979.

3. Appelbe, W. F., R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, Using Mutation Analysis for Testing Ada Programs, *Proceedings of Ada-Europe '88*, Munich, W. Germany June 1988.

4. Bilsel, M. S., *A Survey of Software Test and Evaluation Techniques*, Technical Report GIT-ICS-83/08, School of Information and Computer Science, Georgia Institute of Technology, April 1983.

5. Bradley, G. H., Algorithm and Bound for the Greatest Common Divisor of *n* Integers, *Communications of the ACM* **13**, 7, July 1970, 433-436.

6. Budinger, C. A. and D. S. Guindi, *Mothra User Manual*, Technical Report GIT-SERC-87/11, Software Engineering Research Center, Georgia Institute of Technology, August 1987.

7. Budd, T. A., R. J. Lipton, R. A. DeMillo and F. G. Sayward, *Mutation Analysis*, Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, April 1979.

8. Budd, T. A., *Mutation Analysis of Program Test Data*, Ph.D. thesis, Department of Computer Science, Yale University, 1980.

9. Budd, T. A. and D. Angluin, *Two Notions of Correctness and Their Relation to Testing*, Technical Report 80-19b, Department of Computer Science, University of Arizona, 1980.

10. Budd, T. A., R. A. DeMillo, R. J. Lipton and F. G. Sayward, The design of a prototype mutation system for program testing, *AFIPS National Computer Conference Proceedings*, Vol. 47, 1978, pp. 623-627.

11. Budd, T. A., R. Hess and F. G. Sayward, *EXPER Implementor's Guide*, Department of Computer Science, Yale University, 1980.

12. Clarke, L. A., A System to Generate Test Data and Symbolically Execute Programs, *Transactions on Software Engineering* **2**, 3, September 1976.

13. Clarke, L. A. and D. J. Richardson, The Application of Error-Sensitive Testing Strategies to Debugging, *Symposium on High-Level Debugging*, ACM SIGSOFT/SIGPLAN, March 1983, 45-52.

14. DeMillo, R. A., D. E. Hocking and M. J. Merritt, *A Comparison of Some Reliable Test Data Generation Procedures*, Technical Report GIT-ICS-81/08, School of Information and Computer Science, Georgia Institute of Technology, April 1981.

15. DeMillo, R. A., D. Guindi, K. N. King, E. W. Krauser, W. M. McCracken, A. J. Offutt and E. H. Spafford, *Mothra Internal Documentation*, Version 1.0, Technical Report GIT-SERC-87/10, Software Engineering Research Center, Georgia Institute of Technology, 1987.

16. DeMillo, R. A., R. J. Lipton and F. G. Sayward, Hints on test data selection: help for the practicing programmer, *Computer* **11**, 4 April 1978, 34-41.

17. DeMillo, R. Lipton, R. J. A. and F. G. Sayward, Program mutation: a new approach to program testing, *Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers*, Infotech International, 1979, pp. 107-126.

18. DeMillo, R. A., W. M. McCracken, R. J. Martin and J. F. Passafiume, *Software Testing and Evaluation*, Benjamin/Cummings, Menlo Park, California, 1987.

19. DeMillo, R. A. and E. H. Spafford, *The MOTHRA Software Testing Environment*, Technical Report GIT-SERC-87/01, Software Engineering Research Center, Georgia Institute of Technology, January 1987.

20. Foster, K., Error Sensitive Test Case Analysis, *IEEE Transactions of Software Engineering* **6**, 3, May 1980.

21. Geller, M., Test Data as an Aid in Proving Program Correctness, *Communications of the ACM* **21**, 5, May 1978.

22. Scheifler, R.W., Gettys, J., *The X Window System*, ACM Transactions on Graphics #63, 1986.

23. Girgis, M. R. and M. R. Woodward, An experimental comparison of the error exposing ability of program testing criteria, *Workshop on Software Testing Conference Proceedings*, IEEE Computer Society Press, July 1986, pp. 51-60.

24. Hanks, J. M., *Testing Cobol Programs by Mutation: Volume I - Introduction to the CMS.1 System, Volume II - CMS.1 System Documentation*, Technical Report GIT-ICS-80/04, School of Information and Computer Science, Georgia Institute of Technology, 1980.

25. Hoare, C. A. R., Proof of a Program: FIND, *Communications of the ACM* **14**, 1, January 1971.

26. Howden, W. E., Reliability of the path analysis testing strategy, *IEEE Transactions on Software Engineering* **SE-2**, 3, 1976 208-215.

27. Howden, W. E., Functional testing and design abstractions, *Journal of Systems and Software* **1**, 4, 1980, 307-313.

28. Howden, W. E., Weak Mutation Testing and Completeness of Test Sets, *IEEE Transactions on Software Engineering* **8**, 2, July 1982.

29. Kernighan, B. W. and P. J. Plauger, *The Elements of Programming Style*, Second Edition, McGraw-Hill, New York, 1978.

30. King, J. C., Symbolic execution and program testing, *Communications of the ACM* **19**, 7, 1976, 385-394.

31. Krauser, E. W. and A. P. Mathur, Program Testing on a Massively Parallel Transputer Based System. *Proceedings of the ISMM International Symposium on Mini and Microcomputers and their Applications*, Austin, Texas, November 1986.

32. Krauser, E. W., A. P. Mathur and V. Rego, High Performance Testing on SIMD Machines, *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, July 1988.

33. Offutt, A. J. and K. N. King, A Fortran 77 interpreter for mutation analysis. *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, June 1987. Published as *SIGPLAN Notices* **22**, 7, July 1987, 177-188.

34. Ramamoorthy, C. V., S. F. Ho and W. T. Chen, On the Automated Generation of Program Test Data, *IEEE Transactions on Software Engineering* **2**, 4, December 1976.

35. Richardson, D. J. and M. C. Thompson, The RELAY Model for Error Detection and its Application, *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, July 1988.

36. Tanaka, A., *Equivalence Testing for Fortran Mutation System Using Data Flow Analysis*, M.S. thesis, School of Information and Computer Science, Georgia Institute of Technology, 1981.

37. Walsh, P. J., *A Measure of Test Case Completeness*, PhD dissertation, Watson School of Engineering, State University of New York at Binghamton, 1985.

38. Woodward, M. R. and K. Halewood, From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, July 1988.

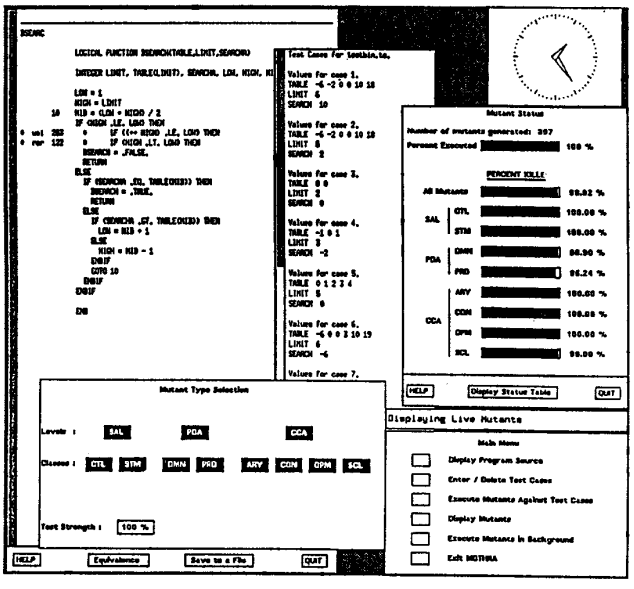

Figure 6. Mothra Display During Testing