

A Fortran Language System for Mutation-Based Software Testing^{† ‡}

K. N. KING

*Department of Mathematics and Computer Science, Georgia State University, Atlanta, GA 30303,
U.S.A.*

AND

A. JEFFERSON OFFUTT

Department of Computer Science, Clemson University, Clemson, SC 29634, U.S.A.

SUMMARY

Mutation analysis is a powerful technique for testing software systems. The Mothra software testing project uses mutation analysis as the basis for an integrated software testing environment. Mutation analysis requires executing many slightly differing versions of the same program to evaluate the quality of the data used to test the program. The current version of Mothra includes a complete language system that translates a program to be tested into intermediate code so that it and its mutated versions can be executed by an interpreter.

In this paper, we discuss some of the unique requirements of a language system used in a mutation-based testing environment. We then describe how these requirements affected the design and implementation of the Fortran 77 version of the Mothra system. We also describe the intermediate language used by Mothra and the features of the language system that are needed for software testing. The appendices contain a full description of the intermediate language and the mutation operators used by Mothra.

The design and implementation techniques that were developed for Mothra are applicable for constructing not just software testing systems, but any type of program analysis system or language system for a special-purpose application. In particular, we discuss decisions made and techniques developed by the Mothra team that can be useful in such applications as debuggers, program measurement tools, software development environments, and other types of program analysis systems.

KEY WORDS Fortran Intermediate code Interpreters Mothra Mutation analysis
Software testing

[†]Research supported in part by Contract F30602-85-C-0255 through Rome Air Development Center. The bulk of this work was done while the authors were with the Georgia Institute of Technology.

[‡]A preliminary version of this paper appeared in the proceedings of the ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques, St. Paul MN, June 1987.

THE MOTHRA MUTATION SYSTEM

The Mothra testing project was initiated in 1986 by members of the Georgia Institute of Technology's Software Engineering Research Center [1]. Mothra is a complete, flexible software test environment that supports mutation-based testing of software systems. It was implemented in the C programming language under the Ultrix-32 operating system and has been ported to a variety of BSD and System V UNIX environments. Mothra was designed as a collection of "plug-compatible" tools based on shared data structures that are stored as files and treated as abstract objects. This design has allowed Mothra to evolve to a remarkable degree as a growing group of researchers continues to add new tools and capabilities, implement different user interfaces that allow for novel styles of interaction, and modify the system for special-purpose experimentation.

At the core of this collection of tools is a set of programs and objects that enable Mothra to translate, execute, and modify programs. We refer to this portion of Mothra as the language system. The language system must satisfy several unusual requirements. In this paper we describe how these unusual requirements were met and the rationale behind some of the more novel design decisions. The first part of the paper lists some of the requirements of the Mothra system and discusses Mothra's overall architecture. We next introduce the major language-related subsystems of Mothra and the data structures that they share. We then discuss how each language-related subsystem is implemented—focusing on those aspects that are different from the "standard" ways of building language systems.

We hope this paper provides valuable insights for building language systems for special-purpose applications. In particular, we discuss techniques that can be useful in program analysis systems such as debuggers, testing systems, and development environments.

Mutation analysis

Techniques for generating sets of test data include path coverage [2], symbolic testing [3], functional testing [4], as well as mutation analysis [5, 6]. (The references [7, 8] survey these and other techniques.) Because software testing cannot guarantee program correctness [2], these techniques

do not attempt to establish the absolute correctness of a program but to provide the tester with some level of confidence in the program. Although each of these techniques is effective for detecting errors in programs, mutation analysis goes one step further by supplying the user with information in the absence of errors. This unique ability helps the tester predict the reliability of a program and indicates quantitatively when the testing process can end. Furthermore, mutation analysis has been shown analytically and experimentally to be a generalization of other test methodologies [9, 10]. 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 testing.

Mutation analysis [5, 11] helps the user create test data and then interacts with the user to improve the quality of that test data. Mutation analysis involves constructing a set of *mutants* of the test program, each of which is a version of the test program that differs from the original by one *mutation*. A mutation is a single syntactic change that is made to a program statement (generally inducing a typical programming error). Figure 1 shows a simple Fortran function containing a mutated statement (preceded by the \rightsquigarrow symbol).

```

FUNCTION MAX (M,N)
MAX = M
IF (N .GT. M) MAX = N
 $\rightsquigarrow$  IF (N. LT. M) MAX = N
END

```

Figure 1: *Function MAX*

The mutant version of MAX is created by replacing the original IF statement by a mutated statement in which the relational operator GT is replaced by LT. Note that this change actually creates a different program.

A program is mutated by applying a *mutation operator* to it. The mutation operators currently supported by the Mothra system can be divided into three broad classes:

- *Statement analysis (sal)*: replace each statement by TRAP (an instruction that causes the program to halt, killing the mutant); replace each statement by CONTINUE; replace each statement in a subprogram by RETURN; replace the target label in each GOTO statement; replace the label in each DO statement.

- *Predicate analysis (pda)*: take the absolute value and negative absolute value of expressions; replace each arithmetic operator by all others; replace each relational operator by all others; replace each logical operators by all others; insert unary operators preceding expressions; alter the values of constants; alter DATA statements.
- *Coincidental correctness (cca)*: replace scalar variables, array references, and constants by other scalar variables, array references, and constants; replace references to array names by the names of other arrays.

These operators were derived from studies of programmer errors and correspond to simple errors that programmers typically make. This particular set of mutation operators [12] represents more than ten years of refinement through several mutation systems. The operators in this set not only require that the test data meet statement and branch coverage criteria, extremal values criteria, and domain perturbation, but also directly model many types of errors. For example, the coincidental correctness operators represent cases in which the programmer uses the wrong variable name or array reference. The predicate analysis operators represent errors that programmers make inside expressions—using an incorrect comparison operator or the wrong arithmetic operator, for example. The statement analysis operators check for several types of errors. The TRAP replacement ensures that each statement is reached, the CONTINUE replacement ensures that each statement is necessary for correct execution of the program, and the RETURN replacement ensures that the code following that statement is necessary. The label replacements, of course, emulate the error of having used the wrong label.

The complete set of mutation operators used by the Mothra mutation system is shown in Table 1. Each of the 22 mutation operators is represented to by a three-letter acronym. For example, the “array reference for array reference replacement” (*aar*) mutation operator causes each array reference in a program to be replaced by each other distinct array reference in the program.

Test cases are used to *kill* mutant programs by differentiating the output of the mutants from that of the original program. If a test case causes a mutant program to produce incorrect output, then the test case is strong enough to detect the fault(s) represented by that mutant and

Type	Description	Class
aar	array reference for array reference replacement	cca
abs	absolute value insertion	pda
acr	array reference for constant replacement	cca
aor	arithmetic operator replacement	cca
asr	array reference for scalar variable replacement	cca
car	constant for array reference replacement	cca
cnr	comparable array name replacement	cca
crp	constant replacement	pda
csr	constant for scalar variable replacement	cca
der	DO statement end replacement	sal
dsa	DATA statement alterations	pda
glr	GOTO label replacement	sal
lcr	logical connector replacement	pda
ror	relational operator replacement	pda
rsr	RETURN statement replacement	sal
san	statement analysis (replacement by TRAP)	sal
sar	scalar variable for array reference replacement	cca
scr	scalar for constant replacement	cca
sdl	statement deletion	sal
src	source constant replacement	cca
svr	scalar variable replacement	cca
uoi	unary operator insertion	pda

Table 1: *Mothra mutation operators*

the mutant is considered dead. The goal when using mutation analysis is to construct a set of test cases powerful enough to kill a large number of mutant programs. Each set of test cases is used to compute an *adequacy score*; a score of 100% indicates that the test cases kill all mutants of the program. Of course, we can never kill all mutants, since some are functionally equivalent to the original test program.

Mutation language system requirements

Mutation testing systems generally create large numbers of mutant programs. For example, Mothra generates 970 mutants for a particular 27-line program. Although the most obvious way to create a mutant program is to create a copy of the original program source, change it, and then compile and execute the mutants, this approach is too computationally and spatially expensive, because the entire test program would have to be stored and compiled thousands of times. Furthermore, it is difficult to use the source program for locating expressions and statements to mutate.

These problems suggest the use of incremental compilation. Incremental compilation could save much of the expense of parsing each mutant, and remove the necessity of storing the complete mutant programs, but still allow execution of mutants at the speed of a compiled program. Despite its theoretical attractiveness, we decided against incremental compilation for the initial version of Mothra because the technology was undeveloped and there were few off-the-shelf incremental compilers available.

Instead, we chose a compromise between the two techniques—Mothra translates the original test program into an intermediate form, applies the mutation operators to change this intermediate code, and then interprets the resulting instructions. Using intermediate code has the advantage of requiring the expensive translation step to be done only once and allowing mutants to be represented as small changes to the intermediate code (usually a replacement of one intermediate code instruction). A further advantage is that Mothra is able to exercise greater control over the execution of mutant programs than if compiled code were used. On the other hand, an interpreted program cannot execute at the same speed as a compiled program.

With this initial decision in mind, Mothra's language system had to satisfy the following requirements:

1. The intermediate code must be easy to mutate.
2. The testing system must closely monitor the input/output behavior of the program under test.
3. It must be possible to decompile the intermediate code to display both the original Fortran program and each mutant program.
4. The test program must not cause the testing system to fail.
5. The interpreter must execute large numbers of mutated programs; thus, the interpreter's speed must be optimized when possible.
6. The interpreter must handle a substantial subset of Fortran 77.

In the following paragraphs, we discuss these requirements in more detail.

1. *Mutate code.*

The intermediate code must allow efficient application of the mutation operators. Although mutation operators are defined at the source code level, they are actually applied to intermediate code. Consequently, there must be an efficient way to detect the type of source language statement from which an intermediate code instruction was derived. Also, the system must have an efficient way to modify the intermediate code.

2. *Monitor input/output behavior.*

The result of testing a program with mutation analysis is based on the input supplied to a program and the output generated by that program. The interpreter must be able to store and repeatedly use input presented to the program as well as capture and examine the program's output. Since the tester may ask to begin execution at an arbitrary subroutine or function, values for input parameters must be placed in the program's memory space before execution begins. Because the program being tested may be used interactively, the values typed by the user during original execution must be saved and used during mutant execution.

3. *Support reverse compilation.*

To design test cases that kill individual mutants, the tester needs to view them. Thus, a mutation system must be able to display mutations as syntactically correct statements. Because mutations are applied to intermediate code, it must be possible to decompile this code into Fortran statements.

4. *Handle errors.*

During mutant execution, testers are actively looking for errors in the program. Rather than handling errors as exceptional behavior, the interpreter must treat them as normal, expected events. This is of particular importance to mutation analysis because the purpose of executing a mutant is not to provide the tester with answers but to provide output to compare with that of the original program. This means that an error condition is not an exception, but a "value" to be used in this comparison. Most importantly, the interpreter must protect against the program under test

causing the test system itself to fail. Failures in the test program must be trapped and used to kill mutants.

5. *Optimize interpreter speeds.*

Because the interpreter is executed during the inner loop of the mutation process and the parser is not, the parser should perform as much computation as possible to allow the interpreter to execute faster. Unfortunately, this requirement is often at odds with other requirements, decomposition and mutant creation in particular.

6. *Handle a substantial subset of Fortran.*

For Mothra to be useful to test nontrivial programs, the language system must handle a large subset of Fortran. At present, the only major feature not completely implemented is input/output; only READ and PRINT statements are supported. The other missing features are statement functions, alternate returns from subroutines, entry points, assumed-size array arguments, and subprograms used as arguments.

MOTHRA ARCHITECTURE

Mothra is a set of tools that can be called separately or through one of the Mothra interfaces [11, 12]. This section describes the goals of this architecture and the approaches used to meet these goals. We then discuss the tools involved in the Mothra language system and how they communicate through files. Later in the paper, each tool is described in more detail. A complete technical description of the entire Mothra system can be found in the Mothra Internal Documentation [13].

Goals

The implementation of the Mothra language system was guided by several goals. First, Mothra was to build on previous work. Several prototype mutation systems were studied during the design and implementation of Mothra. The EXPER system [14] proved to be particularly helpful. The designers of EXPER had already faced the problems of designing a language system for mutation analysis; we based much of Mothra's interpreter on the algorithms and data structures

used by EXPER. We also considered reusing existing code to be an important goal of the project. The construction of the first version of Mothra in a relatively short time was made possible by relying on the UNIX *f77* libraries for I/O and intrinsic functions. Another goal of our interpreter was to provide test harness capabilities. Because testing is often performed on subprograms rather than on an entire program, a testing system must serve as a test harness [7] to provide the test subprogram with appropriate values for its parameters and global variables. This goal is reflected in the input/output and the program failure protection requirements.

Mothra architectural approach

The basic approach to building Mothra was to implement each major function as a separate program that executes independently of other tools. This approach provides several advantages that are important when building software research systems. Specifically, it allows the Mothra system to be “plug-compatible” with other tools by allowing the addition of new programs to the current Mothra tool set without affecting existing tools. This separability also enhances Mothra’s role as a prototype experimental system by making changes and updates easy to implement. Mothra’s tool set approach requires that tools communicate through files. Each file is accessed through independent code that allows tools to view the file as an abstract object. The data structures can be modified without affecting the Mothra tools; in the future, these files can be replaced by more powerful databases.

The tools included in Mothra’s language system are the *parser*, the mutant maker (*mutmake*), the test case formatter (*mapper*), the interpreter (*rosetta*), and the decoder (*decode*). Other tools within Mothra perform such functions as status reporting and test case generation or provide interfaces to the Mothra tools. Before describing each tool in depth, let us consider how the system is used and how the tools interact (see Figure 2).

When supplied with a program to test, Mothra first invokes the parser to create an intermediate code file and a symbol file. The code file contains instructions in *Mothra intermediate code* (MIC), a postfix intermediate language. The symbol file contains tables of program names and their attributes.

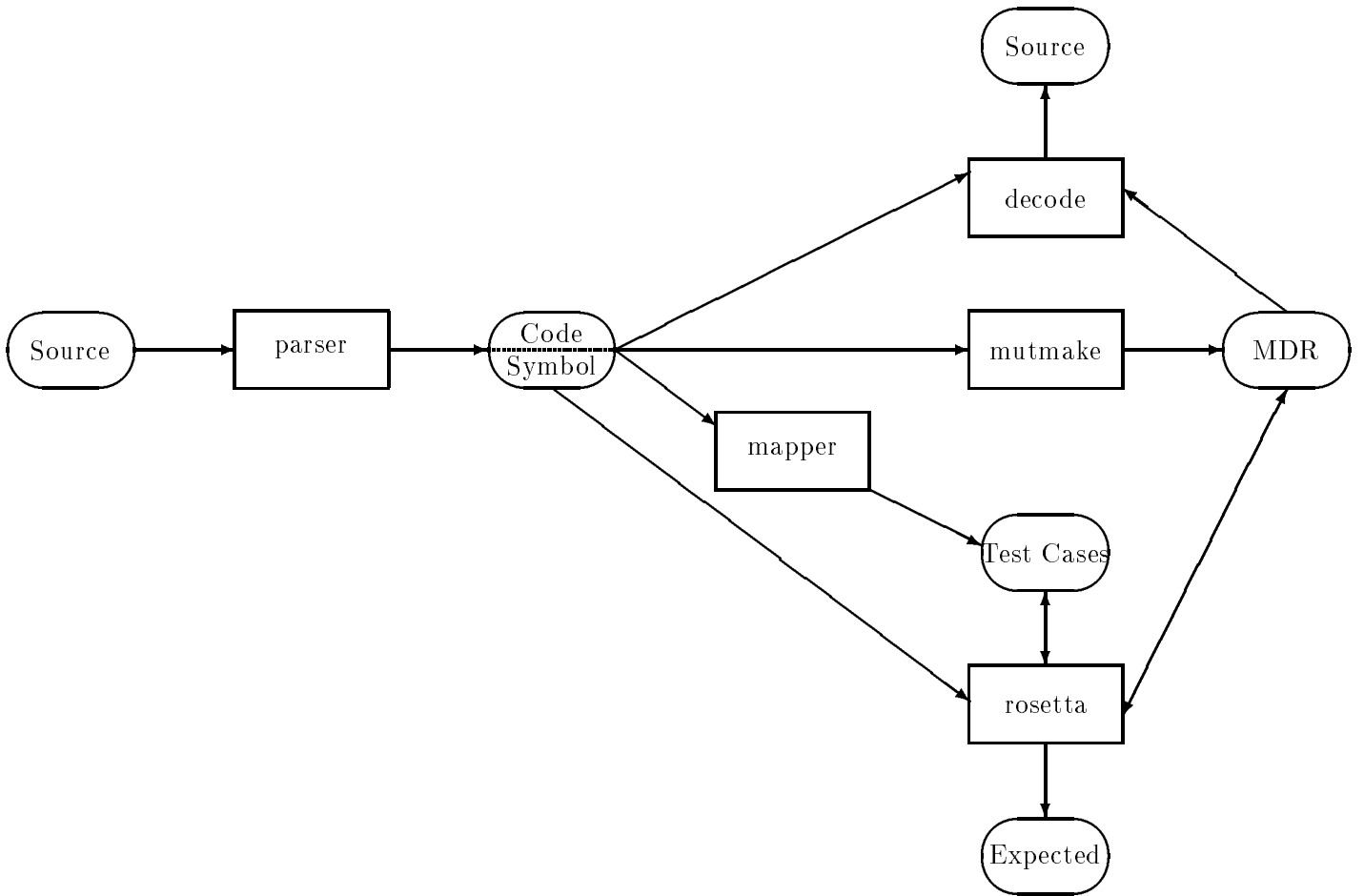


Figure 2: *Mothra language system architecture*

After parsing the program, Mothra invokes *mutmake* to produce a file of *mutant descriptor records* (MDRs). Each record describes the change(s) to the code file needed to produce a single mutant. The number of mutants for a program is on the order of the square of the number of names in the program. Using MDRs avoids the cost of storing many slightly differing copies of the same code. MDRs are applied to the code file at run time to create ready-to-execute mutated versions of the test program.

Before executing these mutants, the tester must submit a test case that the Mothra system uses to produce the original output. Test cases are entered by interacting with *mapper*¹. When given a test case, Mothra invokes *rosetta* to execute the original version of the code file (*original execution*) and save the output of the test program for later comparison. Mothra also saves the test case for use during execution of mutants (*mutant execution*). After original execution, the tester must verify that the output of the original program on that test case is correct.

¹There are currently four ways of generating test cases; using previous test cases, using *mapper*, using a spreadsheet [12], or generating test cases automatically [15]. Using *mapper* is the simplest method and the most relevant for this paper.

Once a test case and a set of MDRs has been created, Mothra executes each mutant program. For each MDR, *rosetta* applies the changes described in the MDR to the MIC table to produce an (in-memory) mutated version of the code. The interpreter then executes the mutated program. The resultant mutant output is compared to the original (correct) output. If the outputs differ, *rosetta* marks the mutant dead; dead mutants are not executed against subsequent test cases. If the outputs are the same, the mutant is left alive. The loop in which *rosetta* generates, executes, and evaluates each mutant in turn is called the *inner loop* because it is the most computationally expensive part of the mutation process. Because the MDR file may contain thousands of records, the speed of *rosetta* is critical to the functioning of Mothra.

Mothra tools

Each Mothra tool implements a major function of mutation analysis. We now describe the capabilities of these tools. Later sections discuss the implementation of the tools.

parser translates a Fortran 77 program into MIC instructions, which are stored in the code file. *parser* also creates a symbol file.

mutmake is responsible for generating mutant descriptor records from the information in the code and symbol files. *mutmake* steps through the MIC instructions and, at each instruction, uses rules to generate the appropriate mutations of that instruction. Although these mutations are defined at the source level, *mutmake* actually follows rules that directly modify MIC instructions. *mutmake* is usually able to avoid generating mutated programs that would be illegal under the rules of Fortran 77. Expressions, however, cause problems, because changing the type of an operand anywhere in an expression can change the type of the entire expression. As a result, type checking and type conversion is done during interpretation.

mapper interacts with the tester to create test cases. Some variables are supplied with values before execution begins. *mapper* prompts the user to supply values for these variables and formats them into a test case. Variables that are initialized during execution, perhaps through READ statements, are not given initial values.

rosetta executes MIC instructions. It accepts as input a code file, a symbol file, a test case

file and, optionally, an MDR file. Depending on the options used, *rosetta* executes the original program or mutant programs on one or more test cases.

decode is a general-purpose tool used to view Fortran programs and related information. *decode* allows the tester to examine the entire program, a single program unit, or a single line. Information that can be displayed includes source lines, MIC instructions generated for these lines, and mutated versions of these lines. Symbol table information may also be viewed. The behavior of *decode* is controlled by options that can be combined to produce varying amounts of information.

Shared data structures

Mothra's principal data structures are implemented as independent objects. Access to each data structure (and the file in which it is stored) is controlled by self-contained code that is compiled into any tool that needs the structure. This independence promotes sharing between the Mothra tools as well as orthogonality in the design. During use, all data structures are stored in memory as dynamically allocated arrays. If, during execution, a data structure grows beyond its current bounds, its array is automatically reallocated (using the C *realloc* function) to be twice its previous size.

The key data structures are the symbol table, the statement and code tables, the data code and data value tables, the test case stream, and the MDR file. Mothra stores each of these data structures as a separate file. The tester supplies Mothra with a global "experiment name", which is used as a base name for each of the files.

When a Fortran program is parsed, a code file and a symbol file are created. The symbol file contains the program's **symbol table**, which is used by *mutmake*, *mapper*, *rosetta*, and *decode* (and most other Mothra tools). Internally, the symbol table consists of several smaller tables:

- a table containing type records,
- a table containing information about local names (e.g., variables, arguments, and symbolic constants),
- a table containing global names (e.g., program units and common blocks), and

- a table that relates internal numbers assigned to program units to the order in which the units appear in the source file.

The table of local names contains information about all local names in a program; the local names belonging to a particular program unit occupy contiguous entries in this table.

The **statement table** contains one entry for each executable statement in the source program. The statement table is built by the parser and used during execution to differentiate between executing Fortran statements and MIC instructions. Each entry in the statement table contains fields for the statement label (if any), a *nesting level* for the statement, and the index of the first MIC instruction generated for the statement. The INTERPRETATION section explains the meaning of the nesting level and discusses how the statement and code tables are used during execution.

The **code table** contains one entry for each MIC instruction. Each record in the code table contains an opcode and a (possibly null) operand. The *mutmake*, *rosetta* and *decode* tools all use the code table.

Fortran DATA statements require special handling by the parser, interpreter, and mutant maker. DATA statements are translated into MIC instructions and stored in the **data code table**. Values appearing in DATA statements are stored in a parallel table called the **data value table**. Before executing the instructions belonging to a program, *rosetta* initializes memory by executing the instructions in the program's data code table. These instructions fetch values from the data value table and then store them into memory. This mechanism allows DATA statements to be mutated in the same way as executable statements.

A test case consists of a sequence of values to be placed into memory before execution begins plus any values supplied to READ statements during original execution. The initial values are stored into the **test case stream** by *mapper*. During original execution, the user responds to READ statements by interactively supplying input values. To ensure that mutants of the original program are executed with the same values, these run-time inputs are saved and reused in the order they were originally supplied. Because the sequence of READ statements in a mutant program may not match that of the original program, a mutant may read values of an incorrect type. Since the

usual result is an output error, causing the mutant to be marked dead, the types of the run-time inputs are not kept and the values are placed on a linked list to form a stream of test case values.

The **MDR file** contains one record for each mutant. Each record contains the information necessary to modify the code file to create the mutated program. For example, the MDR for the mutant shown in Figure 1 would indicate the code table index for the change, the mutation type (a relational operator replacement), and the new operand to be substituted into the code table at that point. The MDRs also contain fields for status information such as whether the mutant is live, dead, or equivalent, and which test case killed it. Because of its size, the Mothra tools access this file directly from disk by reading short segments of the file into memory. Our object-oriented design hides this complication from the tools by encapsulating it in the module that accesses the MDRs.

PARSING

Mothra's *parser* was designed to satisfy two goals that differ from those of a normal compiler. Because the output of *parser* will be mutated, the code generated must contain enough information about the source program to enable *mutmake* to generate mutants that correspond to legal Fortran programs. Furthermore, decompiling a mutated program must be possible, a second reason for retaining source information in the generated code. At the same time, the code generated must allow efficient interpretation (not just of the original program, but its mutated versions as well).

parser is a modified version of the first pass of *f77*, the Fortran compiler supplied with BSD 4.3 UNIX. Only the lexical analyzer and parser are retained from *f77*. We decided not to use the *f77* symbol table, because the symbol table needed for a mutation system must retain more information and for longer periods and must be accessed in many different ways. The only symbol table structure retained from *f77* is the label table, which is used only during parsing.

We used the lexical analyzer from *f77* without change. The *f77* parser was generated by *yacc*, the UNIX parser generator. The *f77 yacc* productions, with minor changes, are the basis for

parser. To create *parser*, we first commented out all semantic actions in the productions. We then gradually added new semantic actions to implement an increasing number of Fortran features.

For simplicity, *parser* operates in one pass, generating the MIC instructions for each construct as the construct is parsed. This works well except in a few cases, notably forward jumps, implied DO loops, and DATA statements, which require backpatching.

parser assumes that it will be given a valid Fortran 77 program as input; consequently, it performs little error-checking. Most error messages are lexical (produced by the lexical analyzer) or syntactical (detected by the *yacc*-generated portion of the parser); few semantic error messages are produced.

Mothra intermediate code

MIC instructions are simple operator-operand pairs. Although mutations are defined at the source language level, they are applied to MIC instructions to save the expense of generating intermediate code for every mutant program. Because of this fact and the decompiling requirement, MIC instructions were designed so that the code produced for a program would have a direct relationship with the original source. This causes MIC to be higher-level than most intermediate forms. For example, a DO statement is not translated to tests and branches but instead represented by a special MIC instruction. Thus, the Fortran statements

```
DO 10 I = 1, 5
    SUM = SUM + I
10 CONTINUE
```

are translated into the MIC instructions shown in Figure 3.

The IDDOLOOP and IDNOCONST opcodes are variants of the IDENT opcode. The IDDOLOOP opcode indicates that I is used as the index variable of a DO loop and therefore should not be replaced by a constant or array element during mutation. The IDNOCONST opcode indicates that SUM appears on the left side of an assignment statement and therefore should not be replaced by a constant. (All three opcodes are treated identically by *rosetta*.) The third CONST instruction gives the increment of the DO loop; the DO statement did not specify the increment, so the parser supplied it.

IDDOLOOP	I
CONST	1
CONST	5
CONST	1
DOSTMT	10
IDNOCONST	SUM
IDENT	SUM
IDENT	I
BINARYOP	OPPLUS
ASSIGN	
CONT	

Figure 3: *DO loop MIC instructions*

The high-level form of MIC allows easy decompilation and mutation at the expense of some run-time efficiency. For example, the DOSTMT opcode makes it easy to locate and replace label references during mutation but requires extra work during interpretation.

Appendix I lists the complete MIC instruction set.

CREATING MUTANTS

When given an experiment name and a list of mutation operators, *mutmake* generates MDRs and writes them to the MDR file for the experiment. On request, *mutmake* is capable of enabling only a percentage of the MDRs or just the MDRs for individual program units.

To reduce the number of MDRs generated, *mutmake* attempts to suppress mutants that are equivalent to the original program or equivalent to some other mutant. To avoid generating these superfluous mutants, *mutmake* uses an *expression stack*. As it scans the sequence of MIC instructions that belong to an expression, *mutmake* uses the expression stack to evaluate the properties of the expression (and its subexpressions). Properties of an expression include whether it is negative, positive, or zero, and whether it is even or odd. These properties can always be determined for constant expressions and can often be determined for nonconstant expressions (for example, $I ** 2$ is always nonnegative, regardless of the value of I). One use of these properties is for suppressing absolute value mutations that would be equivalent to the original program. For example, *mutmake*

does not generate $ABS(I * *2)$ from $I * *2$.

The expression stack also allows *mutmake* to locate operators and operands belonging to a particular expression. *mutmake* can, at any point during the processing of an expression, call one of four functions: **fetch_result**, **fetch_left_operand**, **fetch_right_operand**, or **find_operator**. The first three fetch information about the topmost expression on the stack; **fetch_result** returns the known properties of this expression, while **fetch_left_operand** and **fetch_right_operand** return the known properties of the operands that appeared in the expression (assuming it involved a binary operator). **find_operator** can be called immediately after *mutmake* has scanned an operand; it locates the operator that will eventually use the operand and, if the operator is binary, returns the properties of the other operand. In effect, **fetch_left_operand** and **fetch_right_operand** allow *mutmake* to go “down” one level into an expression that has already been processed, while **find_operator** allows *mutmake* to go “up” one level into an expression that has not yet been completely processed. These functions enable *mutmake* to avoid generating mutants that match a particular “pattern”.

Appendix II lists the mutation operators that Mothra uses and describes the special cases that *mutmake* handles.

HANDLING INPUT AND OUTPUT

The principal purpose of mutation analysis is to help the user create test data that differentiates the behavior of a program from slight variations of that program. A test case kills a mutant only if the test case causes the mutant to produce different (incorrect) output from the original program on the same test case. For each test case, Mothra must store both the input used during original execution and the output generated.

During unit and module testing, testers often want to test software with inputs that are not taken from READ statements and to base correctness on values that are not displayed with PRINT statements. Thus, Mothra allows certain variables to be given values before execution and allows

the final values of certain variables to be treated as the output of the program. Each variable in the first subprogram that is executed has a *class* attribute. There are four classes: initial (**IN**) indicates that the variable should get a value before execution begins; output (**OUT**) indicates that the variable's final value should be considered part of the program's output; initial/output (**INOUT**) combines both **IN** and **OUT**; and don't care (**DONTCARE**) indicates that neither the initial nor the final value of the variable is relevant to the correctness of the program.

The determination of a variable's class depends on the initial subprogram that *rosetta* executes. If there is a main program, the tester must specify the class of each of its variables. If the initial subprogram is a Fortran subroutine, then the tester defines the class of each parameter. If the tester defines all the variables to be **DONTCARE**, then the program is assumed to handle all input/output on its own (through `READ` and `PRINT` statements²). Any variables appearing in `COMMON` blocks are defined to be **INOUT**. If the initial subprogram is a function, the value of the function is assigned the class **OUT**.

During original execution, the final values of **OUT** and **INOUT** variables are saved in a file. During mutant execution, the final values of these **OUT** and **INOUT** variables are compared against the values saved from the original execution. If the program contains Fortran output statements, the values printed by these statements are also captured and saved during original execution and compared against the output of the mutated code. If the initial subprogram is a function, then the value that it returns is also compared.

A test case for a program consists of all the input values read by the program during original execution. **IN** and **INOUT** variables are assigned values before execution begins from the test case stream set up by *mapper*. Interactive input, however, is more difficult. *rosetta* handles interactive input during original execution by using the BSD 4.3 UNIX Fortran 77 I/O library, trapping the sequence of bytes that are read and storing them with the current test case. These run-time input values are read in the same order during mutant execution they were entered during original execution. During mutant execution, the values that were read during original execution

²Because there is no clear way to fit references to external files into this model, file I/O is not supported by Mothra. Previous mutation systems did not handle I/O at all.

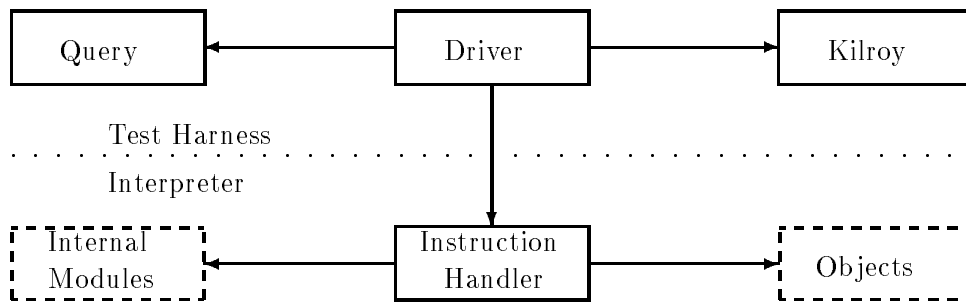


Figure 4: *Rosetta architecture*

are carefully placed into memory to mimic the effect of a Fortran input statement.

INTERPRETATION

Mothra’s interpreter, *rosetta*, executes the original and mutant programs. Besides executing MIC instructions, *rosetta* handles multiple test cases and/or multiple mutants, kills mutants, and includes a test harness. Figure ?? shows the major components of *rosetta*; the arrows indicate the calling hierarchy.

The top level of *rosetta* (the portion above the dotted line in Figure ??) implements the test harness functions. Initially, *rosetta* determines how many test cases are to be executed (this can be controlled via command line arguments) and begins a loop over each test case (the test case loop). If *rosetta* is executing mutant programs, it sets up a loop to execute each mutant (the mutant loop), otherwise it executes the (original) code file once. Within the test case loop (and mutant loop if present), *rosetta* accesses the test case stream and places the initial values in memory using the Query module. If the program is to begin execution at a subroutine rather than a main program, *rosetta* initializes the execution stack and program counter to emulate a call of that subroutine.

After using the Instruction Handler module to execute the code file, *rosetta* performs one of two operations. If it is executing the original version of the program, *rosetta* uses the results of PRINT statements and the values of **OUT** and **INOUT** variables to create the expected output file. If it is executing a mutant program, *rosetta* compares the mutant’s output against the original output to see if the mutant should be marked dead using the Kilroy module.

rosetta has several interesting capabilities that are unusual in an interpreter. The next six subsections describe some of these features.

Original versus mutant execution

Most of the time, *rosetta* does not know whether it is executing original or mutated code. The portion of *rosetta* represented by the bottom half of Figure ?? behaves almost identically in either case. At some points, however, principally at the test harness level and during READ statements, *rosetta* behaves differently depending on its execution mode. When in original execution mode, *rosetta* reads the symbol table, code table, and test case stream, creates the output file and “tags” file (discussed later), and modifies the test case stream. During mutant execution, *rosetta* reads the symbol table, code table, test case stream, MDR file, output file, and tags file, and updates the MDR file.

During original execution, *rosetta* executes the original program just once. Upon encountering a READ statement, *rosetta* queries the test case stream for a value if there is one, otherwise it accepts input values from standard input (usually the keyboard) and adds them to the test case stream. After execution completes, the test case stream is written to the test case file. The output of PRINT statements and the final values of **OUT** and **INOUT** variables are written to standard output, which is saved in the output file. The tags file stores a “stop code” and a record of which statements within the test program were executed.

During mutant execution, *rosetta* generates and executes each live mutant that alters a statement that was executed in the original program by the current test case. Mutants are generated by changing the in-memory version of the code table. Upon encountering a READ statement, *rosetta* will obtain a value from the test case stream if one is available. If there are no values left, *rosetta* will use “null” values (zero for numeric types, blanks for characters, etc.) for the remaining READ statements. As in original execution, standard output is diverted, but this time to a direct comparison with the data in the original output file. Finally, *rosetta* compares the original output with the mutant output and if they differ, marks the mutant as being killed. This comparison is currently done on a byte-by-byte basis. Since the comparison does not allow for different, yet

still correct mutant behavior (for example, negligible differences in real number values), current research is examining ways of performing a less restrictive comparison.

Initialization

Before *rosetta* begins executing a program, it performs several actions that are crucial to its capabilities as a test harness. These initialization steps allow the test program to execute as if it were being executed in the intended environment, rather than inside a test system. In this section, we discuss the most interesting aspects of initialization.

One common difference between machines is the contents of memory before execution begins. Although some machines initialize a program's memory space to zeros before execution begins, others leave previous data in memory. This affects a program's behavior when references are made to uninitialized variables or to invalid array subscripts. Both cases are important in mutation-based testing because mutation operators can replace initialized variables with uninitialized variables or change valid array subscripts to subscripts that are out of range.

To cope with this problem, *rosetta* can perform a memory initialization step before execution begins. If this option is requested, *rosetta* is called with a value to be used for memory initialization. *rosetta* uses a large array of bytes to represent memory; each program variable is assigned the correct number of bytes for its type³. During the memory initialization step, each byte in memory is filled with the specified value.

Software is often tested at the module level rather than at the program level, so *rosetta* can begin execution inside an arbitrary subprogram. If there is a main program, *rosetta* executes the entire program. If not, *rosetta* begins execution with the first subprogram in the source file. The top level of *rosetta* simulates a subprogram call so that the lack of a main program is transparent to the instruction handler. *rosetta* first sets up space in memory for any parameters (if there is no call to the subprogram, the parser did not allow space for the actual values of the parameters). Then the address of this space is placed into the formal parameter's memory space, just as in a normal subprogram call. Finally, a special return address is placed onto the stack so that when

³Since Fortran 77 does not support recursion or dynamic storage allocation, memory can be allocated statically.

the subprogram is finished executing, *rosetta* can recognize that the program should terminate normally. If the subprogram is a function, space is set aside for the function's return value so that it can be used as part of the program's output.

If the program being tested does not contain a main program, the parameters and global variables used by the initial subprogram need values before execution. For each **IN** or **INOUT** variable a value is taken from the test case stream and placed into memory as if it had been assigned via a Fortran assignment statement.

Execution loop

Another novel feature of *rosetta* is the structure of its main execution loop. To allow testing to proceed as rapidly as possible, *rosetta* differentiates between source statements and individual MIC instructions. The main execution loop in *rosetta* operates on two distinct levels, the *statement level* and the *code level*. Thus, *rosetta*'s program counter actually consists of two different counters, the *statement pointer* and the *code pointer*. These counters are incremented separately, but the code pointer always indexes a MIC instruction that is within the statement indexed by the statement pointer. The separate statement-level loop allows *rosetta* to perform various bookkeeping activities when statements are executed.

The major action performed at the statement level is DO statement checking. Figure 3 gives an example of the MIC instructions generated for a DO loop. This code contains all the information necessary for mutation and decompiling. The code does not indicate where the DO loop ends, however, forcing *rosetta* to find the end on its own. *rosetta* uses statement *nesting level* information to accomplish this task efficiently. As the program is parsed, the level of DO-loop nesting is recorded in the statement table. During execution, if the current statement is nested at a lower level than the previous statement, then the program has left a DO loop; the interpreter needs to increment the loop counter and either go to the next loop iteration or terminate the loop. *rosetta*'s two-level execution loop minimizes the number of these DO loop checks.

Because mutating a program may cause an infinite loop, *rosetta* protects the mutation system by checking for infinite loops. During original execution, *rosetta* records the number of statements

executed. During mutant execution, this count is compared against the current statement count. If the current statement count exceeds the original count (multiplied by a user-supplied constant), then the mutant is killed. For efficiency, this counting and comparing is performed at the statement level.

The TRAP mutation operator is also handled at the statement level. The TRAP operator replaces each line in the program by a special TRAP statement. Execution of a TRAP statement immediately kills the mutant. During optimization, *rosetta* was modified so that TRAP mutants are not actually executed. Instead, a table is built during original execution to keep track of which statements were executed. During mutant execution, each TRAP mutation whose affected statement appears in the table is marked dead without being executed.

Handling failures

rosetta performs extensive checks on all instructions executed. If an error is found, then an error code is propagated back to the execution loop and execution is halted immediately. This *stop code* indicates why execution stopped and becomes part of the program's output state. This extensive error checking is a computational expense that is necessary to maintain the integrity of the Mothra testing system.

rosetta is capable of detecting several kinds of errors in a program. Some of these are errors introduced during mutation. (Care is taken during mutant generation not to produce illegal programs, but certain kinds of errors are difficult to detect at this stage.) Others are more conventional error conditions that are detected by most interpreters. The errors that *rosetta* detects include:

- *Bad type.* An expression contains an illegal combination of types. This error is sometimes introduced during mutation.
- *Bad assignment.* An assignment involving incompatible type is detected. This error also results from mutation.
- *Time out.* A mutated program is taking too long to execute, possibly indicating an infinite loop. Detection of this error was discussed earlier.

- *Subscript out of range.* Because subscript checking is not required in Fortran 77, *rosetta* does not perform subscript checking unless requested to do so.
- *Out of memory bounds.* Mutations involving array references can access locations outside the bounds of *rosetta*'s internal memory array. To protect the system from these mutations, accesses outside this array are detected and treated as failures.

Killing mutants

After executing a mutant program, *rosetta* compares the output of the mutant with the original output of the same test case. First *rosetta* compares the two stop codes. If they differ, then *rosetta* immediately marks the mutant dead. Otherwise, *rosetta* does a byte-by-byte comparison of the output stream of the original and mutated programs. If the mutant is killed, the MDR record for that mutant is updated; on subsequent test cases, that MDR record is not used to generate an executable mutant.

Tracing

rosetta contains a tracing feature that has proved useful not only to test *rosetta* but also to test *parser*, *mutmake* and the definition of the mutants themselves. We patterned the trace after the UNIX communications program *uucp* [16]. *rosetta* has four levels of trace that give increasing levels of detail in the output. The information produced at each level includes all the information from lower levels. The four levels currently implemented are:

1. Extra status information is printed after execution stops.
2. Source statements are displayed as they are executed. (This feature uses the Mothra decompiler.)
3. The result of each assignment statement is shown.
4. MIC instructions are printed as they are executed.

Level 1 tracing is handled in *rosetta*'s main driver, level 2 tracing is handled in the statement loop, and the last two levels are handled in the instruction loop. Thus, higher trace levels cause a definite performance degradation as well as a marked increase in output.

One of the more surprising uses for the tracing feature is in creating test cases. Because tracing can display the statements that are executed and even the values assigned during execution by a test case, tracing is useful for understanding why test cases do not kill mutants. This simple feature has become an important part of the process of using Mothra.

DECOMPILING

decode displays information about a Fortran program, including the original source program, information about the symbols in the program, the code generated for the program, and mutations of the program. *decode*'s numerous options can be combined in many ways, making it a powerful, general-purpose tool.

decode can display various regions of a program: either a single line, a single program unit, or all program units (the entire program). Alternatively, individual mutants can be displayed. This option causes the original source line and mutated line to be displayed. However, no other information can be shown when this option is selected.

Along another dimension, *decode* allows various types of information to be displayed for the selected program region. *decode* can display the original source lines, the MIC instructions, the decompiled source lines (created by decompiling the MIC instructions), and/or the mutations of those lines (live, dead, and/or equivalent). When the selected region is a program unit or the entire program, the symbol table or data table for the selected unit(s) can be displayed.

One of the most interesting aspects of *decode* is the “bottom-up” method it uses to decompile a mutated Fortran statement from a sequence of MIC instructions. As it reads the instructions, *decode* constructs character strings containing fragments of the statement and saves these on a stack. From time to time, *decode* pops small fragments from the stack, joins them together, and

then pushes the new string back onto the stack. (*decode* mimics the push and pop operations of *rosetta*, except that it creates a string representing each operation rather than performing the operation itself.) After *decode* has processed the last MIC instruction belonging to the statement, it prints all strings on the stack.

For example, suppose that *decode* is about to decompile the statement corresponding to the following MIC instructions:

```
IDENT      I
CONST     2
IDENT      I
CONST     1
BINARYOP  +
BINARYOP  *
ASSIGN
```

decode first pushes “I” onto the stack, then “2”, then “I”, and finally “1”. When it encounters the first BINARYOP opcode, *decode* pops “1” and “I” and pushes the string “(I + 1)”. (Expressions involving operators are always kept fully parenthesized while on the stack.) The second BINARYOP opcode causes *decode* to pop “(I + 1)” and “2” and push “(2 * (I + 1))”. When *decode* encounters the ASSIGN opcode, it pops “(2 * (I + 1))” (removing the outer level of parentheses), pops “I”, and builds the string “I = 2 * (I + 1)”, which it pushes on the stack. *decode* now prints the contents of the stack.

Because *decode* may work with relatively long strings (statements continued over several lines), it is important—for the sake of efficiency—to avoid unnecessary string copying. *decode* accomplishes this goal by working exclusively with pointers to strings. When it needs room to hold a statement fragment, *decode* dynamically allocates space for a buffer long enough to contain the longest possible statement. When *decode* pushes a string onto the stack or pops it off the stack, only a pointer is actually copied (the stack contains pointers to strings, not strings themselves). When a string buffer is no longer needed, *decode* releases its space.

PERFORMANCE

rosetta contains the inner execution loop of the Mothra mutation system; it is responsible for executing and evaluating large numbers of mutant programs without interaction from the tester. *rosetta* is by far the most CPU-intensive portion of Mothra, and its performance has a critical impact on Mothra's usability.

We used an object-oriented methodology [17] to design *rosetta*. A common argument against object-oriented design is the inefficiency of accessing data structures through procedure calls. However, object-oriented design actually provided some straightforward ways to optimize interpretation. Using abstract data types for memory, the stack, the symbol table, and the code table provided a convenient interface between *rosetta* and the other Mothra tools. This design greatly simplified the optimization stage, which consisted initially of eliminating redundant code and isolating critical code for detailed optimization. It also allowed us to locate critical algorithms to be rewritten for greater efficiency. In the current system, the overhead of an increased number of procedure calls has been reduced by replacing selected procedure calls by macro invocations.

CURRENT AND FUTURE WORK

Implementing the initial version of Mothra's language system required approximately six man-months. Since 1986, it has been used by researchers, practitioners, and educators at dozens of sites and demonstrated with several versions of the Mothra interface. Mothra has been undergoing maintenance for the same period of time and, despite its size (initially about 50,000 lines, currently almost 100,000), has repeatedly been found to be easy to understand, improve, modify, and fix. Mothra has executed Fortran programs of up to 1000 lines and currently satisfies all the requirements mentioned earlier in this paper.

Since the initial language system was finished, Mothra has been extended in a number of ways. A system for automatically generating test data has been implemented; it reuses the front

end of *mutmake* to transform the code table and the MDRs into logical constraints on the program variables, which are then used to generate input values to kill mutants [15]. Other new tools include a program that transforms MIC into a dataflow graph to detect equivalent mutants [18], a spreadsheet tool for manipulating test cases [12], and a debugger that incorporates mutation testing information [12]. Currently, we are working on a version of *rosetta* that implements the “weak mutation” approach suggested by Howden [19]. Other efforts include optimizing the internal algorithms used in *rosetta*, optimizing MIC itself, and modifying the architecture of the interpreter.

Split-stream execution

One architectural method currently under study for Mothra’s interpreter takes advantage of properties that are unique to mutation analysis. During mutation analysis, the program is executed many times—once in the original form and once for each mutant. Because each mutant contains only one mutated statement, the execution stream of a mutated program is identical to that of the original version up to the point at which the mutated statement is executed. Executing each mutated program from the beginning results in duplicated work. A *split-stream* approach reduces this duplication of effort by having the interpreter split the execution stream of the original program to begin mutant execution at the point where the mutated statement appears. The execution paths form a tree like the one in Figure ?? (Boxes represent mutated program lines and circles represent original source lines.)

If properly implemented, this technique could significantly improve the execution speed of a mutation interpreter. Unfortunately, the overhead of keeping track of so many different states of the program makes this scheme difficult to implement on a single-processor machine. The technique of split-stream execution is being used in algorithms for mutation-based testing on parallel machines, however [20,21].

Data flow

Looking at the interpreter’s execution of a program from the perspective of dataflow techniques gives another method for speeding up mutant execution. A flow graph that represents

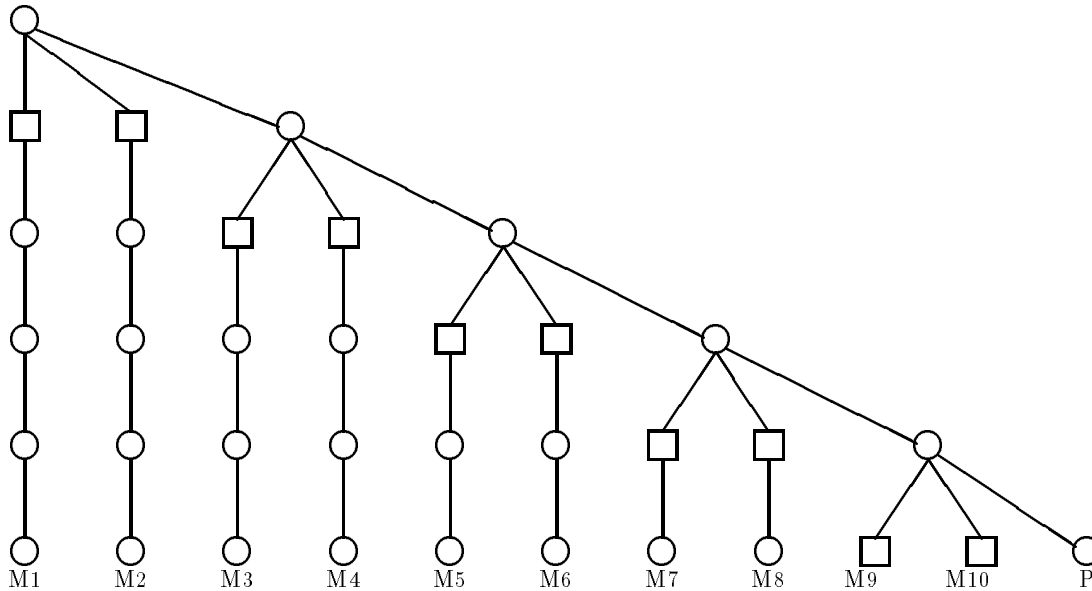


Figure 5: *Split-stream execution*

changes in the program’s data space can be constructed from the original execution of the test program [22]. Because each mutation affects only one node in this flow graph, an interpreter could avoid executing an entire mutated program by making the appropriate change to the affected node and propagating the change throughout the flow graph to generate the final output. This technique is similar to Howden’s weak mutation [19] but less restrictive.

Multiple source languages

During the current phase of Mothra’s development, we are generalizing the system to handle multiple source languages. This step requires extensions to MIC, revisions to the tools, an updated set of mutation operators, and the addition of parsers for the other source languages, including Ada and C.

CONCLUSIONS

The Mothra software testing environment consists of an extensive tool set [12]. With help and guidance from an advanced user interface, a tester can specify testing goals, automatically generate

test cases to satisfy test criteria, execute the program and determine input/output pair correctness or equivalence of mutants, manipulate or fine tune the test cases, and debug the program when errors are revealed.

These capabilities require extensive support from the underlying language system. We have found Mothra intermediate code to be simple and efficient. MIC instructions have been used for interpretation, various types of symbolic analysis [23], data flow analysis [18], decompilation, automatic generation of test data [24], and are currently being used to develop a debugger [12]. There is much information stored in the MIC instructions and in the Mothra symbol table, yet the information is simple to understand and easy to access. This combination of power and simplicity is important to integrated program analysis systems and other research software that is expected to grow over time. As bottlenecks in the testing process shift in response to new capabilities, additional tools will be built and integrated into Mothra, allowing it to serve as an experimental software testing vehicle for years to come.

The design and implementation techniques the Mothra team developed to satisfy our particular goals and requirements are useful in applications other than mutation analysis and software testing. The way we designed and implemented the Mothra language system has been instrumental in the continuing success of the software. We expect these techniques to be useful for building large-scale program analysis systems, research software, and educational tools.

ACKNOWLEDGEMENTS

A large and diverse team contributed to the original implementation of Mothra by designing, writing code, testing, and advising. We would like to acknowledge the efforts of Richard DeMillo, Ed Krauser, Ronnie Martin, Mike McCracken, and Gene Spafford. In addition, we would like to acknowledge the many researchers who have contributed to Mothra since the original implementation was finished, including Hira Agrawal, Mike Craft, William Hsu, Wynne Hsu, Aditya Mathur, Hsin Pan, and Jason Seaman.

APPENDIX I: MOTHRA INTERMEDIATE CODE

This appendix describes the Mothra Intermediate Code (MIC) language that is produced by the parser and executed by the interpreter. The first table lists each operator in the MIC language, a short description of that operator, the operands that operator can use, and a description of the action the interpreter takes. The abbreviations, symbols, and constructs that appear in the following tables are:

- `pop`—Remove and return the value of the element on top of the expression stack.
- `push(x)`—Push x onto the expression stack.
- `NULL`—No operand needed or action taken.
- `sp`—The current value of the statement pointer.
- `cp`—The current value of the code pointer.
- `m[x]`—The memory location of the variable x .
- `op`—The operand of this operator.
- `code(cp)`—Returns the operand field of the MIC instruction at index cp .
- `‡`—This instruction was included for mutation purposes and is generated by *mutmake* rather than by the parser.
- `⇒`—Look to the operand table for the operands that are used by this operator.

Mothra Intermediate Code Instruction Set

Operator Table

Operator	Description	Operand	Action
Expression operators			
BINARYOP	Binary operator	Binary operands \Rightarrow	push($op(pop_2, pop_1)$)
UNARYOP	Unary operator	Unary operands \Rightarrow	push($op(pop)$)
ARRAYOP	Array operator	Array operands \Rightarrow	pop;pop;Compute addr;push(addr)
SETBOUNDS	Flexible array operator	Number of dimensions	Define array dimensions
BIFNCT	Built-in function operator	Intrinsic functions	push($op(pop_n, \dots, pop_1)$)
INSERTOP†	Insertion operator	NULL	NULL
ASSIGN	Assignment	NULL	m[pop] \leftarrow pop
ASSIGNTEMP	Assignment to temporary variable	Temporary variable	m[op] \leftarrow pop
ASSIGNDATA	Assigning during a DATA statement	NULL	m[pop] \leftarrow constant
String operators			
CONCAT	Concatenate two strings	NULL	push(pop pop)
SUBSTR	Take a substring	NULL	push(pop(pop : pop))
Branching operators			
TRFALSE	Transfer if false	Statement index	if (pop = 0) sp \leftarrow op
ARITHIF	Arithmetic IF	NULL	if (pop < 0) sp \leftarrow code(cp+1) if (pop = 0) sp \leftarrow code(cp+2) if (pop > 0) sp \leftarrow code(cp+3)
CGOTO	Computed GOTO	Number of labels	sp \leftarrow code(cp+pop)
AGOTO	Assigned GOTO	Number of labels	sp \leftarrow pop
SKIP	Skip several code instructions	Number to skip	cp \leftarrow cp + op
BRANCH	Unconditional branch to a statement	Statement index	sp \leftarrow op
TRANSFER	Nonmutable branch to a statement	Statement index	sp \leftarrow op
JUMP†	Nonmutable jump to a new code line	Code index	cp \leftarrow op
IF	Null action IF statement	NULL	NULL
THEN	Null action THEN statement	NULL	NULL
ENDIF	Null action ENDIF statement	NULL	NULL
ELSE	Null action ELSE statement	NULL	NULL
ELSEIF	Null action ELSEIF statement	NULL	NULL
Flow control operators			
CONT	CONTINUE statement	NULL	NULL
TRAP†	Trap	NULL	Abort program with error
STOP	STOP statement	NULL	Abort program without error
DOSTMT	DO statement	Ending label	Start DO loop
ONETRIP†	One-trip DO statement	Ending label	Execute loop at least once
LABEL	Statement label	Statement index	Push (op)
Subroutine and function operators			
RET	Return from subroutine or function	Return operands \Rightarrow	End subroutine or function call
SUBCALL	Subroutine call	Subprogram index	Start subroutine call
FUNCALL	Function call	Subprogram index	Start function call
PARMFLG	Parameter list start	NULL	push (PARMFLG)

Mothra Intermediate Code Instruction Set

Operator Table (continued)

Operator	Description	Operand	Action
Scalar operators			
IDENT	Identifier	Symbol table index	push(<i>op</i>)
IMMUTIDENT	Identifier that cannot be mutated	Symbol table index	push (<i>op</i>)
IDNOCONST	Identifier cannot be replaced by const	Symbol table index	push (<i>op</i>)
IDDOLOOP	Identifier used as a DO loop index	Symbol table index	push (<i>op</i>)
CONST	Generic constant	Constant index	push(constant)
IMMUTCONST	Constant that cannot be mutated	Integer constant	push(<i>op</i>)
Input/output operators			
STARTREAD	Begin a READ statement	IO operands \Rightarrow	Begin input of form <i>op</i>
ENDREAD	Finish a READ statement	IO operands \Rightarrow	End input of form <i>op</i>
STARTWRITE	Begin a PRINT statement	IO operands \Rightarrow	Begin output of form <i>op</i>
ENDWRITE	Finish a PRINT statement	IO operands \Rightarrow	End output of form <i>op</i>
DOIO	Perform I/O with one element	IO operands \Rightarrow	Read or write one element
FORMAT	A format string	String format index	push(<i>op</i>)
Implied DO loop operators			
IMPDOTEST	Implied DO loop start	Code index	Start implied DO loop
IMPDOINIT	Implied DO loop end	Code index	End implied DO loop
Internal interpreter operator			
MICEXPR	Expression marker	NULL	NULL
Mutation replacement operators			
Operator	Description	Operand	Action
STARTREPL†	Start of a code replacement	Codeindex	$cp \leftarrow op$
ENDREPL†	End of a code replacement	Codeindex	$cp \leftarrow op$

The following table lists the operands that are used by the operators in the previous table.

Mothra Intermediate Code Instruction Set	
Operand Table	
Operand	Meaning
Binary operands (BINARYOP)	
OPPLUS	$X + Y$
OPMINUS	$X - Y$
OPSTAR	$X * Y$
OPSLASH	X / Y
OPPOWER	$X ** Y$
OPMOD†	$rem(X / Y)$
OPLEFTVAL†	$X \langle op \rangle Y \rightarrow X$
OPRIGHTVAL†	$X \langle op \rangle Y \rightarrow Y$
OPOR	$X \text{ or } Y$
OPAND	$X \text{ and } Y$
OPEQV	$X = Y$
OPNEQV	$X \neq Y$
OPTRUE†	Expression is true
OPFALSE†	Expression is false
OPLT	$X < Y$
OPEQ	$X = Y$
OPGT	$X > Y$
OPLE	$X \leq Y$
OPNE	$X \neq Y$
OPGE	$X \geq Y$
Unary operands (UNARYOP)	
OPNEG	$-X$
OPNOT	$\neg X$
OPABS	$abs(X)$
OPNEGABS	$-abs(X)$
OPZPUSH†	Error if expression is 0
OPINC†	$X = X + 1$
OPDEC†	$X = X - 1$
Array operands (ARRAYOP)	
OPLPAREN	Start the indexing
OPCOMMA	Start the next index
OPRPAREN	End indexing, compute final address
Return operands (RET)	
OPNOEXPR	Return with no value
OPEXPR	Place a return value on stack first

Mothra Intermediate Code Instruction Set	
Operand Table (continued)	
Operand	Meaning
Input/Output operands (STARTREAD, ENDREAD, STARTWRITE, and ENDWRITE)	
SEQ_LST_EXT	Sequential, list-directed, external
SEQ_FOR_EXT	Sequential, formatted, external
DIR_FOR_EXT	Direct, formatted, external
DIR_UNF_EXT	Direct, unformatted, external
SEQ_UNF_EXT	Sequential, unformatted, external
SEQ_LST_INT	Sequential, list-directed, internal
SEQ_FOR_INT	Sequential, formatted, internal
Input Output operands (DOIO)	
FOR	Formatted
LST	List-directed
UNFOR	Unformatted

Mothra also handles all Fortran 77 intrinsic functions, which are used by the BIFNCT operator.

APPENDIX II: MOTHRA MUTATION OPERATORS

This appendix describes the twenty-two mutation operators used in Mothra. In the following, x and y represent arbitrary Fortran expressions. References to an expression being positive, nonnegative, negative, nonpositive, or zero mean either that the expression is constant (hence its properties are known) or it is non-constant but its properties can be deduced. For each mutation operator, we describe its semantics, and list special cases in which the operator will not be applied. These exceptions represent attempts to suppress mutations that are equivalent to the original program, illegal, or equivalent to another mutation.

Values of type COMPLEX are not used as replacements in certain contexts (as an array subscript, in assigned GOTO statements, in DO loops, in arithmetic IF statements, in relational expressions, and in computed GOTO statements).

AAR—Array Reference for Array Reference Replacement

Each array reference in a program unit is replaced by every array reference of compatible

type appearing in the program unit. All arithmetic types are considered compatible; character strings are not compatible, even if they have the same length.

Restriction:

1. An array reference is not replaced by itself.

ABS—Absolute Value Insertion

Each arithmetic expression (and subexpression) is preceded by the unary operators ABS, NEGABS, and ZPUSH. ABS computes the absolute value of the expression; NEGABS computes the negative of the absolute value. ZPUSH tests whether the expression is zero. If so, the mutant is killed; otherwise, execution continues and the value of the expression is unchanged.

Restrictions:

1. None of the operators are applied to an expression that is to be immediately tested for equality or inequality with zero. (ABS and NEGABS mutations would be equivalent to the original program. ZPUSH is not needed since the program's behavior will vary depending on whether or not the expression is zero.)
2. None of the operators are applied to an expression whose absolute value is to be computed. (ABS and NEGABS mutations would be equivalent to the original program. ZPUSH would generate a mutation equivalent to another insertion of ZPUSH; e.g., ABS(ZPUSH(I)) is equivalent to ZPUSH(ABS(I)).)
3. None of the operators are applied to the expression in an arithmetic IF statement. (ABS and NEGABS mutations would be equivalent to GLR mutations. ZPUSH is not needed since the program's behavior will vary depending on whether or not the expression is zero.)
4. None of the operators are applied to an expression that is known to be zero, negative, or positive. (ABS and NEGABS mutations would be either equivalent to the original program or equivalent to UOI mutations; e.g., ABS(5) is equivalent to 5, while ABS(-5) is equivalent to

- – 5. The ZPUSH mutation would be either equivalent to the original program or equivalent to a SAN mutation; e.g., ZPUSH(5) is equivalent to 5; ZPUSH(0) is equivalent to TRAP.)
- 5. ABS and NEGABS are not applied to a binary expression of the form $x * y$ or x/y if either x or y is known to be positive, nonnegative, negative, nonpositive, or zero. (ABS and NEGABS mutations would be equivalent to another insertion of ABS or NEGABS; e.g., $\text{ABS}(x * y)$ is equivalent to $x * \text{ABS}(y)$ if x is known to be positive or $x * \text{NEGABS}(y)$ if x is known to be negative.)
- 6. ABS and NEGABS are not applied to an expression that is known to be nonnegative or nonpositive. (The mutations would be either equivalent to the original program or equivalent to UOI mutations; see restriction (4).)
- 7. ABS and NEGABS are not applied to an expression that is to be raised to an even power. (Both mutants would be equivalent to the original program.)
- 8. ZPUSH is not applied to an expression of the form $x * y$, x/y , or $x ** y$. (The mutation would be equivalent to another insertion of ZPUSH or, in the case of $x * y$, equivalent to a combination of two mutations created by inserting ZPUSH; e.g., a test case that kills $\text{ZPUSH}(x * y)$ will kill either $\text{ZPUSH}(x) * y$ or $x * \text{ZPUSH}(y)$.)

ACR—Array Reference for Constant Replacement

Each constant in a program unit is replaced by every array reference of compatible type in the unit. All arithmetic types are considered compatible; character strings are not compatible, even if they have the same length.

AOR—Arithmetic Operator Replacement

Each occurrence of one of the operators $+$, $-$, $*$, $/$, and $**$ is replaced by each of the other operators. In addition, each is replaced by the operators LEFTOP, RIGHTOP, and MOD. LEFTOP returns the left operand (the right is ignored); RIGHTOP returns the right operand. MOD computes the remainder when the left operand is divided by the right.

Restrictions:

1. An operator is not replaced by itself.
2. Arithmetic operator replacement is not performed when so doing would create one of the following mutations: $x + 0$, $0 + x$, $x - 0$, $x * 0$, $0 * x$, $x * 1$, $1 * x$, $x/1$, $x ** 1$, $x \text{ MOD } 1$ (all are equivalent to other AOR mutations), $x/0$, or $x \text{ MOD } 0$ (equivalent to SAN mutations).
3. RIGHTOP and LEFTOP are not used as replacements on the right side of assignment statements when so doing would cause the two sides to become identical. (This would be equivalent to an SDL mutation; e.g., $I = I \text{ LEFTOP } J$ is the same as $I = I$, which is equivalent to CONTINUE.)

ASR—Array Reference for Scalar Variable Replacement

Each scalar variable in a program unit is replaced by every array reference of compatible type in the unit. All arithmetic types are considered compatible; character strings are not compatible, even if they have the same length.

Restriction:

1. A variable used as the index in a DO statement is not replaced. (This replacement would create an illegal Fortran program.)

CAR—Constant for Array Reference Replacement

Each array reference in a program unit is replaced by each constant visible in the unit, provided that the base type of the array and the type of the constant are both arithmetic.

Restrictions:

1. An array reference on the left side of an assignment statement is not replaced by a constant.
2. Constant replacement is not performed when so doing would create one of the following mutations: $x + 0$, $0 + x$, $x - 0$, $x * 1$, $1 * x$, $x/1$, $x ** 1$ (all are equivalent to AOR mutations), or $x/0$ (equivalent to a SAN mutation).

CNR—Comparable Array Name Replacement

In each array reference, the array name is replaced by the name of every other array of compatible type and identical number of dimensions. All arithmetic types are considered compatible; character strings are compatible if they have the same length.

Restriction:

1. An array name is not replaced by itself.

CRP—Constant Replacement

Each constant value is modified slightly to emulate domain perturbation testing. The change to the value depends on the constant's type:

1. Each integer constant is both incremented by 1 and decremented by 1.
2. Each nonzero real and double precision constant is incremented and decremented by 10% of its value; zero is replaced by .01 and $-.01$.
3. Each complex number (r, i) is replaced five times. Each of r and i is mutated separately as a real number; in addition, r and i are interchanged.
4. Each logical constant is replaced by its complement.
5. The first character in each string constant is replaced by its predecessor and successor in the underlying collating sequence. (For example, 'MUTATION' is replaced by 'LUTATION' and 'NUTATION'.)

Restrictions:

1. Constant replacement is not performed when so doing would create one of the following expressions: $x + 0$, $0 + x$, $x - 0$, $x * 1$, $1 * x$, $x/1$, $x ** 1$. (All are equivalent to AOR mutants; for example, replacing the 1 in $x + 1$ to make $x + 0$ is equivalent to x LEFTOP 0.)
2. Constant replacement is not performed when so doing would create a mutant of the form $x/0$ (equivalent to a SAN mutant).

CSR—Constant for Scalar Variable Replacement

Each scalar variable in a program unit is replaced by each constant visible in the same unit, provided that the types of the variable and the constant are both arithmetic.

Restrictions:

1. A variable used as the index in a DO statement is not replaced. (This replacement would create an illegal Fortran program.)
2. A variable used as the left side of an assignment statement is not replaced.
3. Variables appearing in READ statements are not replaced.
4. Replacement is not performed when so doing would create one of the following mutations: $x + 0$, $0 + x$, $x - 0$, $x * 1$, $1 * x$, $x/1$, $x ** 1$ (all are equivalent to AOR mutations), or $x/0$ (equivalent to a SAN mutation).

DER—DO Statement End Replacement

The label in each DO statement is replaced by every label in the same program unit. In addition, each DO statement is replaced by a ONETRIP statement. A ONETRIP statement is identical to a DO statement, except that the loop body is always executed at least once.

Restrictions:

1. A label is not replaced by itself.
2. Replacement of a DO label is not performed if the new label appears in the program unit prior to the DO statement. (This replacement would create an illegal Fortran program.)
3. Replacement of a DO label is not performed if it would cause the range of the DO loop to overlap another DO loop or block IF statement. (This replacement would create an illegal Fortran program.)
4. A DO label is not replaced by the label on a GOTO, assigned GOTO, RETURN, STOP, DO, or arithmetic IF statement. (This replacement would create an illegal Fortran program.)

DSA—DATA Statement Alterations

The DSA mutation is identical to the CRP mutation, but affects constants in DATA statements rather than constants in executable statements.

Restrictions:

See the restrictions for the CRP operator.

GLR—GOTO Label Replacement

The labels in unconditional GOTO, computed GOTO, and arithmetic IF statements are replaced by every label in the same program unit.

Restrictions:

1. A label is not replaced by itself.
2. The replacement of a label is not performed if it would cause a branch into the middle of a DO loop or block IF statement. (This replacement would create an illegal Fortran program.)

LCR—Logical Connector Replacement

Each occurrence of one of the logical operators (.AND., .OR., .EQV., .NEQV.) is replaced by each of the other operators; in addition, each is replaced by FALSEOP, TRUEOP, LEFTOP, and RIGHTOP. LEFTOP returns the left operand (the right is ignored); RIGHTOP returns the right operand. FALSEOP always returns .FALSE. and TRUEOP always returns .TRUE..

Restrictions:

1. An operator is not replaced by itself.
2. The main operator in the expression that controls a logical IF statement is not replaced by FALSEOP. (This replacement would be equivalent to an SDL mutation; e.g., IF (.FALSE.) I = 0 is equivalent to CONTINUE.)

ROR—Relational Operator Replacement

Each occurrence of one of the relational operators (.LT., .LE., .GT., .GE., .EQ., .NE.) is replaced by each of the other operators and by FALSEOP and TRUEOP. FALSEOP always returns .FALSE. and TRUEOP always returns .TRUE..

Restrictions:

1. An operator is not replaced by itself.
2. The main operator in the expression that controls a logical IF statement is not replaced by FALSEOP. (This replacement would be equivalent to an SDL mutation; e.g., IF (.FALSE.) I = 0 is equivalent to CONTINUE.)
3. Operators in complex expressions are replaced only by .EQ. and .NE. and by FALSEOP and TRUEOP.

RSR—RETURN Statement Replacement

Each statement in a subroutine or external function (including the statement inside a logical IF) is replaced by RETURN.

Restrictions:

1. RETURN statements are not replaced.
2. END statements are not replaced. (The resulting mutant would be equivalent to the original program.)
3. A GOTO statement whose target is a RETURN statement is not replaced. (The resulting mutant would be equivalent to the original program.)

SAN—Statement Analysis

Each statement at the beginning of a basic block and each statement inside a logical IF statement is replaced by TRAP. TRAP is a special mutation statement; any mutant executing a TRAP statement is immediately killed. The first statement in a basic block can be any of the following:

- The first statement in a program unit.
- Any labeled statement.
- Any ELSEIF, ELSE, or ENDIF statement.
- Any statement following an IF (arithmetic, logical, or block), ELSE IF, GOTO (unconditional, computed, or assigned), DO, RETURN, or STOP statement.
- Any statement following the end of a DO loop.

Restriction:

1. An END statement is only replaced by TRAP if it has a label. (It is not an error if an unlabeled END statement is never reached.)

SAR—Scalar Variable for Array Reference Replacement

Each array reference in a program unit is replaced by every scalar variable of compatible type appearing in the program unit. All arithmetic types are considered compatible; character strings are compatible if they have the same length.

Restriction:

1. A variable is not used as a replacement on the right side of an assignment statement when so doing would cause the two sides to become identical. (This replacement would be equivalent to an SDL mutation; e.g., $I = I$ is equivalent to CONTINUE.)

SCR—Scalar for Constant Replacement

Each constant in a program unit is replaced by every scalar variable of compatible type visible in that unit. All arithmetic types are considered compatible; character strings are compatible if they have the same length.

Restriction:

1. A variable is not used as a replacement on the right side of an assignment statement when so doing would cause the two sides to become identical. (This replacement would be equivalent to an SDL mutation; e.g., $I = I$ is equivalent to CONTINUE.)

SDL—Statement Deletion

Each statement is replaced by CONTINUE.

Restrictions:

1. A CONTINUE statement is not replaced by CONTINUE.
2. An END statement is not replaced by CONTINUE. (This replacement would create an illegal Fortran program.)
3. A block IF, ELSE IF, ELSE, or END IF statement is not replaced by CONTINUE. (None of these is really a statement by itself; each is a part of a larger block IF construct.)
4. In an external function or subroutine, a RETURN statement occurring immediately before an END statement is not replaced by CONTINUE. (This mutant would be equivalent to the original program.)
5. In a main program, a STOP statement occurring immediately before an END statement is not replaced by CONTINUE. (This mutant would be equivalent to the original program.)

SRC—Source Constant Replacement

Each arithmetic constant in a program unit is replaced by every arithmetic constant visible in that unit.

Restrictions:

1. A constant is not replaced by itself.
2. An integer constant is not replaced by another integer constant whose value differs by plus or minus one (equivalent to a CRP mutation).

3. Constant replacement is not performed when so doing would create one of the following mutations: $x + 0$, $0 + x$, $x - 0$, $x * 1$, $1 * x$, $x / 1$, $x ** 1$ (all are equivalent to AOR mutations), or $x / 0$ (equivalent to a SAN mutation).

SVR—Scalar Variable Replacement

Each scalar variable in a program unit is replaced by each scalar variable of compatible type visible in the same unit. All arithmetic types are considered compatible; character strings are compatible if they have the same length.

Restrictions:

1. A variable is not replaced by itself.
2. A variable is not used as a replacement on the right side of an assignment statement when so doing would cause the two sides to become identical. (This replacement would be equivalent to an SDL mutation; e.g., $I = I$ is equivalent to CONTINUE.)

UOI—Unary Operator Insertion

Each arithmetic expression is negated, incremented by 1, and decremented by 1. Each logical expression is complemented.

Restrictions:

1. An operand following a binary $+$ or $-$ operator is not negated (equivalent to an AOR mutation; e.g., $I + (-J)$ is equivalent to $I - J$).
2. An expression is not negated if it is to be used in a multiplication or division (equivalent to other UOI mutations; e.g., $(-I) * J$ is equivalent to $-(I * J)$).
3. An expression is not negated if its absolute value is to be computed or it is to be raised to an even power (equivalent to the original program).
4. An expression is not negated if it is to be tested for equality or inequality with zero (equivalent to the original program).

5. An expression is not negated if the next operation is negation (equivalent to other UOI mutations).
6. The constant zero is not negated (equivalent to the original program).
7. In an arithmetic IF statement, the expression is not negated if the first and third labels are the same (equivalent to the original program; e.g., IF (I) 10,20,10 is equivalent to IF (- I) 10,20,10).
8. Integer constants are not incremented or decremented (equivalent to CRP mutations).
9. An expression is not incremented or decremented if it is to be used in an addition or subtraction (equivalent to another UOI mutation; e.g., (++ I) + J is equivalent to ++ (I + J)).
10. The right operand in a relational expression is not incremented or decremented (equivalent to another UOI mutation; e.g., I.LT.(++J) is equivalent to (-I).LT.J).
11. Logical constants are not complemented (equivalent to CRP mutations).
12. Relational expressions are not complemented (equivalent to ROR mutations; e.g., .NOT.(I.LT.J) is equivalent to I.GE.J).
13. A logical expression is not complemented if the next operation is complementation (equivalent to another UOI mutation).

REFERENCES

1. R. A. DeMillo and E. H. Spafford, 'The Mothra software testing environment', *Proceedings of the 11th NASA Software Engineering Laboratory Workshop*, Goddard Space Center, December 1986.
2. W. E. Howden, 'Reliability of the path analysis testing strategy', *IEEE Transactions on Software Engineering* **SE-2**, (3), 208-215 (1976)
3. J. C. King, 'Symbolic execution and program testing', *Communications of the ACM* **19**, (7), 385-394 (1976).
4. W. E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, 1987.
5. R. A. DeMillo, R. J. Lipton and F. G. Sayward, 'Hints on test data selection: help for the practicing programmer', *IEEE Computer* **11**, (4), 34-41 (April 1978).
6. R. A. DeMillo, R. J. Lipton, 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.
7. M. S. Bilsel, '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.
8. R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume, *Software Testing and Evaluation*, Benjamin/Cummings, 1987.
9. T. A. Budd, 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.

10. M. R. Girgis, and M. R. Woodward, 'An experimental comparison of the error exposing ability of program testing criteria', *IEEE Workshop on Software Testing Conference Proceedings*, Banff, Alberta, July 1986, pp. 51-60.
11. Mothra Project Team, 'The Mothra testing environment user's manual', *Technical Report SERC-TR-4-P*, Software Engineering Research Center, Purdue University, September 1987.
12. R. A. DeMillo, E. W. Krauser, R. J. Martin, A. J. Offutt and E. H. Spafford, 'The Mothra tool set', *Proceedings of the Hawaii International Conference on System Sciences*, Kailua-Kona, HI, January 1989.
13. A. H. Agrawal, R. A. DeMillo, Wm. Hsu, Wy. Hsu, K. N. King, E. W. Krauser, A. J. Offutt, H. Pan, and E. H. Spafford, 'Mothra internal documentation, Version 1.5.' *Technical Report SERC*, Software Engineering Research Center, Purdue University, July 1989.
14. T. A. Budd, R. L. Hess, and F. G. Sayward, 'EXPER implementor's guide', *Technical Report*, Department of Computer Science, Yale University, 1980.
15. A. J. Offutt, 'Automatic Test Data Generation', *Ph.D. dissertation (Technical Report GIT-ICS 88/28)*, Georgia Institute of Technology, 1988.
16. D. A. Nowitz and M. E. Lesk, 'A dial-up network of UNIX systems', *UNIX System Manager's Manual*, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, March 1984.
17. G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.
18. W. M. Craft, 'Detecting Equivalent Mutants Using Compiler Optimization Techniques', *Master's thesis*, Department of Computer Science, Clemson University, 1989.
19. W. E. Howden, 'Weak mutation testing and completeness of test sets', *IEEE Transactions on Software Engineering* **SE-8**, (4), 371-379 (1982).

20. E. W. Krauser, A. P. Mathur, and V. Rego, 'High performance testing on SIMD machines', *Proceedings of the IEEE Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988.
21. R. A. DeMillo, E. W. Krauser, and A. P. Mathur, 'Using the hypercube for reliable testing of large software', *Technical Report SERC-TR-24-P*, Software Engineering Research Center, Purdue University, August 1988.
22. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
23. A. J. Offutt, and E. J. Seaman, 'Using symbolic execution to aid automatic test data generation', *Proceedings of the IEEE 1990 Annual Conference on Computer Assurance (COMPASS 90)*, Gaithersburg, MD, June 1990.
24. 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', *Proceedings of the IEEE Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988.