Mutation Analysis Using Mutant Schemata

Roland H. Untch
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
untch@cs.clemson.edu

A. Jefferson Offutt
ISSE Department
George Mason University
Fairfax, VA 22030-4444
ofut@isse.gmu.edu

Mary Jean Harrold
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
harrold@cs.clemson.edu

Abstract

Mutation analysis is a powerful technique for assessing and improving the quality of test data used to unit test software. Unfortunately, current automated mutation analysis systems suffer from severe performance problems. This paper presents a new method for performing mutation analysis that uses *program schemata* to encode all mutants for a program into one *metaprogram*, which is subsequently compiled and run at speeds substantially higher than achieved by previous interpretive systems. Preliminary performance improvements of over 300% are reported. This method has the additional advantages of being easier to implement than interpretive systems, being simpler to port across a wide range of hardware and software platforms, and using the same compiler and run-time support system that is used during development and/or deployment.

Keywords: Fault-based testing, mutation analysis, program schemata, software testing.

1 Introduction

Programs[1] are tested by executing them against test inputs and examining the resulting outputs for errors. The intent of this testing process is to increase our confidence in the correctness of the tested code. However, when testing is poorly conducted, in an ad-hoc manner, our confidence may be misplaced. Poorly selected test data that does not adequately exercise a program must be deemed "low quality". Systematic testing techniques establish *test data adequacy criteria* that seek to measure the quality of the test data used to exercise a given program. One powerful testing technique that uses an adequacy criterion is *mutation testing* [11, 13, 14]. In mutation testing, the test set is analyzed to determine a quality measure called the *mutation adequacy score*; this process is called *mutation analysis*.

Unfortunately, the conventional method of performing mutation analysis, which requires interpreting many slightly differing versions of the same program, has significant problems. Automated mutation analysis systems based on the conventional method are slow, laborious to build, and usually unable to completely emulate the intended operational environment of the software being tested. The principle reason conventional mutation analysis systems are slow is because they are interpretive. As one study noted, "current implementations of mutation tools are unacceptably slow and are only suitable for testing relatively small programs" [16]. Thus, while conventional systems have proved useful for experimentation with mutation testing, the widespread practical use of mutation analysis has been stymied by the enormous computational

[1] We use the word *program* to denote the software under test, which may be a complete program or some smaller unit, such as a procedure.

requirements of these conventional systems. Conventional, interpretive systems are also laborious to build. To test software written in a specific language, interpreter-based systems must incorporate ALL the compilation characteristics and run-time semantics of that language. For certain languages, such as Ada, this is a formidable undertaking. Since dialectical differences often exist, the degree of compliance to language standards becomes a problem. Also, subtle changes in program behavior may occur since the program under test is no longer running in its intended operational environment.

This paper presents a new method of performing mutation analysis that does not suffer from these problems. Rather than mutating an intermediate form of the program that then must be interpreted, our Mutant Schema Generation (MSG) method enables us to encode all mutations into one source-level program. This program is then compiled (once) with the same compiler used during development and is executed in the same operational environment at compiled-program speeds. Since mutation systems based on mutant schemata do not need to provide run-time semantics and environment, they are significantly less complex and easier to build than interpretive systems, as well as more portable.

In section 2 we provide some background on mutation testing. We introduce our Mutant Schema Generation (MSG) method in section 3 and present experimental results in section 4. Related work is reviewed in section 5 and conclusions are presented in section 6.

2 Mutation Testing Background

Mutation analysis is a *white-box* testing technique that is based on the notion that the quality of a test set is related to the ability of that test set to differentiate the program being tested from a set of marginally different, and presumably incorrect, alternate programs. We say that a test case differentiates two programs if it causes the two programs to produce different outputs.

The process of performing mutation analysis on some test set T, relative to a given program P, begins by running P against every test case in T. If the program computes an incorrect result, the test set has fulfilled its obligation and the program must be changed. (Determining the correctness of these results is the "Oracle" problem [24], which is common to all testing techniques and will not be discussed further.)

Assuming P computes correct results for every test case in T, a set of alternate programs is produced. Each alternate program, P_i, known as a *mutant* of P, is formed by modifying a single statement of P according to some predefined modification rule. Such modification rules, G, are called *mutagenic operators* or *mutagens*[2]. The syntactic change itself is called the *mutation*. The original program plus the mutant programs are collectively known as the *program neighborhood*, N, of P [7].

Each mutant is run against the test cases in T. If for some test case in T a mutant produces a result different than that of the original program, we say that test case has "killed" the mutant indicating that the test case is able to detect the faults represented by the mutant. Once killed, these *dead* mutants are not run against any additional test cases.

Some mutants, although syntactically different, are functionally identical to the original program. We call these *equivalent* mutants. Although some progress has been made in automatically identifying which mutants are equivalent [4, 18], this remains a time-consuming manual task. Since no test case can kill these equivalent mutants, they must be removed from consideration in assessing test data quality.

The ratio of dead mutants to the remaining undifferentiated *live* mutants is an indicator of test set quality. In mutation analysis, the measure used to express test set quality is the *mutation adequacy score*, or *MS*. It is the percentage of potentially killable mutants that actually have been killed by T, or

$$MS_G(P,T) = \frac{\#Dead}{\#Mutants - \#Equivalent} \times 100\%$$

where $\#Mutants$ is the total number of mutants in the program neighborhood. We subscript the mutation adequacy score *MS* by the set of mutagens G to reflect their influence on the number and type of mutants produced. In practice, however, a standard set of mutagens is used and it is common for this subscript to be omitted.

The major computational cost of mutation analysis is incurred when running the mutant programs against the test cases. The number of mutants generated for a program is roughly proportional to the number of data references times the number of data

[2] The terminology varies; they are also sometimes called *mutant operators*, *mutation operators*, *mutation transformations*, and *mutation rules* [25]. Acree [1] uses the term *mutagenic operator*; in biology, a mutagenic substance or factor is simply called a *mutagen*.

objects [9], which is typically a large number. For example, 385 mutants get generated for the procedure Newton shown in Figure 1.

```
1    PROCEDURE Newton(Number:REAL; VAR Sqrt:REAL);
2    (* Find square root using Newton's method. *)
3    VAR
4        NewGuess, Delta, Epsilon : REAL;
5    BEGIN
6        Epsilon :=  0.001;
7        NewGuess :=  (Number / 2.0) + 1.0;
8        Sqrt :=  0.0;
9        Delta :=  NewGuess - Sqrt;
10       WHILE Delta > Epsilon DO
11           Sqrt :=  NewGuess;
12           NewGuess := (Sqrt+(Number/Sqrt))/2.0;
13           Delta :=  NewGuess - Sqrt;
14           IF Delta < 0.0 THEN
15               Delta :=  -Delta;
16           END;
17       END; (* END WHILE *)
18   END Newton;
```

Figure 1: Newton square root procedure

3 The MSG Method

Our approach to mutation analysis is based on *program schemata*. A *program schema* is a template. A *partially interpreted program schema*, as defined by Baruch and Katz [5], syntactically resembles a program, but contains free identifiers, called *abstract entities*, in place of some program variables, datatype identifiers, constants, and program statements. A schema is created via a process of *abstraction*. A schema can be *instantiated* to form a complete program by providing appropriate substitutions for the abstract entities.

We have devised a new form of partially interpreted program schema, the *mutant schema* [23]. Mutant schemata are used to represent program neighborhoods. A mutant schema has two components, a *metamutant* and a *metaprocedure set*, both of which are represented by syntactically valid (i.e., compilable) constructs. The use of mutant schemata significantly speeds up mutation analysis.

3.1 Mutation Analysis Using Mutant Schemata

The essence of this new method lies in the creation of a specially parameterized program called the *metamutant*. Derived from the program under test P, the metamutant is compiled using the same standard compiler used to compile P and runs at compiled-speeds. While running, the metamutant functions as any of the alternate programs found in N, the program neighborhood of P.

To explain how a metamutant represents the functionality of a collection of mutants, we must take a closer look at mutation analysis. Recall that for a program P, each mutant of P is formed as a result of a single modification to some statement in P. Thus, each mutant of Newton differs from the original by only one mutated statement. The way in which these statements are altered is dictated by the set G of mutagens (modification rules) used. The discussion below uses the mutagenic operators defined for the IMSCU system [19]; these rules are typical of those in current use [2, 15].

Consider the *arithmetic operator replacement* (AOR) rule, which states that each occurrence of an arithmetic operator is replaced by each of the other possible arithmetic operators. Each operator is also replaced by special operators LEFTOP and RIGHTOP, where LEFTOP returns the left operand (the right is ignored) and RIGHTOP returns the right operand. Applying this rule to the assignment statement of line 9 of Newton

```
Delta := NewGuess - Sqrt;
```

yields the following six mutations:

```
Delta := NewGuess  +         Sqrt;
Delta := NewGuess  *         Sqrt;
Delta := NewGuess  /         Sqrt;
Delta := NewGuess  MOD       Sqrt;
Delta := NewGuess .LEFTOP.   Sqrt;
Delta := NewGuess .RIGHTOP.  Sqrt;
```

These mutations can be "generically" represented as

Delta := NewGuess *ArithOp* Sqrt;

where *ArithOp* is a *metaoperator* abstract entity.

The generic representation above can be recast as a syntactically valid statement

```
Delta := AOrr (NewGuess, Sqrt, 62);
```

where the AOrr function performs one arithmetic operation. (The third argument, "62" in this example, is used to identify the location, or change point, in the program where this function is invoked.) AOrr is an example of a *metaprocedure*, a function that corresponds to an abstract entity in the schema. A statement that has been changed to reflect such a generic form is said to have been *metamutated*. A *metamutation* is a syntactically valid change that embodies other changes.

```
9  Original=>  Delta := NewGuess - Sqrt;
   ---------   ----------------------------------
   [Mutagens] / [Mutations]
   -----       ----------------------------------
   [ABS]       Delta := ABS(NewGuess) - Sqrt;
   [ABS]       Delta := NEGABS(NewGuess) - Sqrt;
   [ABS]       Delta := ZPUSH(NewGuess) - Sqrt;
   [ABS]       Delta := NewGuess - ABS(Sqrt);
   [ABS]       Delta := NewGuess - NEGABS(Sqrt);
   [ABS]       Delta := NewGuess - ZPUSH(Sqrt);
   [ABS]       Delta := ABS((NewGuess - Sqrt));
   [ABS]       Delta := NEGABS((NewGuess - Sqrt));
   [ABS]       Delta := ZPUSH((NewGuess - Sqrt));
   [AOR]       Delta := NewGuess + Sqrt;
   [AOR]       Delta := NewGuess * Sqrt;
   [AOR]       Delta := NewGuess / Sqrt;
   [AOR]       Delta := NewGuess MOD Sqrt;
   [AOR]       Delta := NewGuess .LEFTOP. Sqrt;
   [AOR]       Delta := NewGuess .RIGHTOP. Sqrt;
   [CSR]       Delta := 0.001 - Sqrt;
   [CSR]       Delta := 2 - Sqrt;
   [CSR]       Delta := 1.0 - Sqrt;
   [CSR]       Delta := 0 - Sqrt;
   [CSR]       Delta := NewGuess - 0.001;
   [CSR]       Delta := NewGuess - 2;
   [CSR]       Delta := NewGuess - 1.0;
   [CSR]       Delta := NewGuess - 0;
   [SVR]       Number := NewGuess - Sqrt;
   [SVR]       Sqrt := NewGuess - Sqrt;
   [SVR]       NewGuess := NewGuess - Sqrt;
   [SVR]       Epsilon := NewGuess - Sqrt;
   [SVR]       Delta := Number - Sqrt;
   [SVR]       Delta := Sqrt - Sqrt;
   [SVR]       Delta := Delta - Sqrt;
   [SVR]       Delta := Epsilon - Sqrt;
   [SVR]       Delta := NewGuess - Number;
   [SVR]       Delta := NewGuess - NewGuess;
   [SVR]       Delta := NewGuess - Delta;
   [SVR]       Delta := NewGuess - Epsilon;
   [UOI]       Delta := -NewGuess - Sqrt;
   [UOI]       Delta := ++(NewGuess) - Sqrt;
   [UOI]       Delta := --(NewGuess) - Sqrt;
   [UOI]       Delta := NewGuess - -Sqrt;
   [UOI]       Delta := NewGuess - ++(Sqrt);
   [UOI]       Delta := NewGuess - --(Sqrt);
   [UOI]       Delta := -(NewGuess - Sqrt);
   [UOI]       Delta := ++((NewGuess - Sqrt));
   [UOI]       Delta := --((NewGuess - Sqrt));
```

Figure 2: `Newton` line 9 and its mutations

All mutations produced from applying standard mutagens can be represented by metamutations. Figure 2 shows all the mutations of line 9 that result from applying the complete set of `IMSCU` mutagens. The following statement

```
dr :=
PUTr(OIr(AOrr(OIr(GETr(45),14),OIr(GETr(46),15),3),16),62);
```

embodies all of these alternatives.

When generating the metamutant of P, a list of mutant descriptors, D, is produced. This list details the alternate operations to be used at each change point in the program. Using this list, the metamutant is dynamically instantiated to function as any of the mutants of P. A "driver" or "harness" invokes the metamutant and directs which mutants are to be instantiated. The driver takes care of such administrative matters as managing test case input and output, handling exceptions, comparing mutant output to the original program output, and recording results. The driver also computes and reports statistics about the current status of the mutants, primarily the mutation score. A common driver is used for all metamutants.

Metaprocedures are syntactically valid representations of the abstract entities found in mutant schemata. They can be categorized as either *metaoperators* or *metaoperands*. *Metaoperator* procedures perform one of a class of alternate operations. Each metaoperator is implemented using a case structure. At run-time, a global parameter selects which alternate operation to perform. This parameter's value is set based on information contained in the mutant descriptor list D. The `AOrr` routine is an example of a metaoperator procedure; a (simplified) version of the `AOrr` function is given in Figure 3.

```
PROCEDURE AOrr
  (LeftOp,RightOp:REAL; ChangePt:INTEGER):REAL;
BEGIN
    CASE  Variant(ChangePt)  OF
        aoADD:    RETURN  LeftOp + RightOp;
      | aoSUB:    RETURN  LeftOp - RightOp;
      | aoMULT:   RETURN  LeftOp * RightOp;
      | aoDIV:    RETURN  LeftOp / RightOp;
      | aoMOD:    RETURN  LeftOp MOD RightOp;
      | aoRIGHT:  RETURN  RightOp;
      | aoLEFT:   RETURN  LeftOp;
    ELSE
        Error( "AOrr CASE out of range" );
        RETURN 0.0;
    END;
END AOrr;
```

Figure 3: Simplified version of Arithemetic Operation function.

Creating metaoperator procedures is a straightforward but tedious task. Many metaoperator procedures are identical in form and differ only in the type declarations of their formal parameters and return value. The nature and number of metaoperator procedures needed is a function of

1. G, the set of mutagens,
2. L, the language of P,

and sometimes

3. P, the program.

Metaoperators that depend only on G and L are said to be *intrinsic*. In languages that do not support user-defined types, such as Fortran, all metaoperators are intrinsic and can be generated once, independent of any particular program, and placed

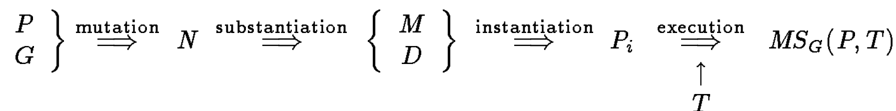

Figure 4: Model of MSG method

in a static metaprocedure library. Metaoperators that depend on G, L, and P are said to be *P-derived*. Only strongly-typed languages, such as Modula-2 or Ada, that allow programs with user-defined types have P-derived metaoperators. For these languages the intrinsic metaprocedure library must be augmented by the P-derived metaoperator routines generated using information from P, the specific program being tested.

Metaoperand procedures reference one of a set of program variables. The actual variable referenced is determined at run-time via a parameter similar to that of the metaoperator procedures. The metaoperand procedures are unique to each program P and must be generated anew for each program. Although the implementation details for metaoperand procedures is influenced by the available features in L, such as scope rules, type checking rules, availability of pointers, and nesting rules, in most cases a scheme using arrays can be used.

A conceptual model of applying the MSG method is given in Figure 4. Working backwards (i.e., from right to left), the mutation adequacy score $MS_G(P,T)$ is obtained as a result of executing the mutants P_i against the test set T. The mutants P_i are obtained by using the list of mutant descriptors D to repeatedly instantiate the metamutant M. M and D are formed as a result of substantiating (i.e., imparting material form to) the program neighborhood N. The program neighborhood is obtained by applying the mutagens G to the program P.

3.2 Generating Metamutants

We are currently developing a complete system to perform mutation testing using mutant schemata. This section gives some details of the system.

The process of generating the metamutant of a program P begins with the construction of a decorated abstract syntax tree. In an *abstract syntax tree* (AST) each non-leaf node represents an operator and the children of the node represent the operands [3]. A *decorated* tree has attributes, such as type information, attached to the nodes. In our system, an attribute grammar is used to direct both the parsing of the program and the AST construction. The resulting AST is decorated with type information by using the symbol table developed during the parsing of the program and semantic rules specified by the attribute grammar. Figure 5 shows a statement and its corresponding AST.

Mutagens are expressed as tree transformation procedures. The mutagens G are applied to the decorated abstract syntax tree. Using the location, type information, and contents of a node and its children, the AST is transformed by replacing node contents with metaprocedure calls. Leaf nodes are replaced by metaoperands and interior nodes are replaced by metaoperators. Each metaprocedure invocation site is a *change point* and is identified by a change point number. Some mutagens cause the structure of the tree to be altered. For example, to accommodate unary operator insertion mutations, the AST is augmented by creating new nodes at certain arcs. Such nodes are represented by hexagons in the rewritten AST on the right hand side of Figure 6. By traversing this revised AST, the information needed to generate a metamutant program is obtained.

4 Experimental Results

As a preliminary step to implementing a full MSG system, we wished to establish empirically that this approach yielded faster mutation analysis than an interpretive approach. We manually generated a metamutant for the Newton procedure shown in Figure 1. An abridged listing of this metamutant is given in Figure 8. We also manually generated a list of mutant descriptors. Using the IMSCU mutagens, 385 mutants were produced. Additionally, a rudimentary library of metaprocedures and a driver were developed and implemented to make a working MSG mutation analysis system.

We compared the speed of the Newton metamutant with that of testing Newton in the Mothra environment—a conventional, interpretive system [15]. A Fortran version of Newton was prepared (see Figure 7) to run under Mothra. Because of slight differences between the IMSCU and Mothra mutagens, there were 364 mutants of the Fortran version of the program.

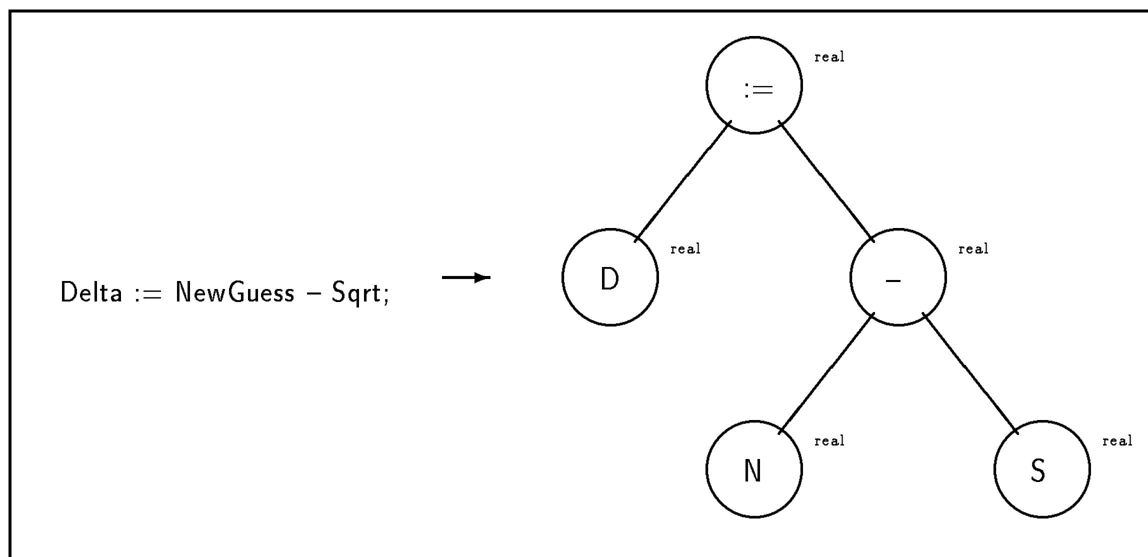

Figure 5: Statement and Corresponding Decorated Abstract Syntax Tree

```
PROCEDURE AOrr
C  Find a square root using Newton's method.
C
C PROCEDURE Newton ( rNum:REAL; VAR Sqrt:REAL);
C                         in              out
      SUBROUTINE Newton(rNum,           Sqrt)
C BEGIN
      Eps =  0.001
      rNew =  rNum / 2.0 + 1.0
      Sqrt =  0.0
      Delta =  rNew - Sqrt
C     WHILE Delta > Epsilon DO
10    IF ( Delta .GT. Eps ) THEN
          Sqrt =  rNew
          rNew =  (Sqrt + rNum/Sqrt) / 2.0
          Delta =  rNew - Sqrt
          IF ( Delta .LE. 0.0 ) THEN
              Delta =  -Delta
          ENDIF
          GOTO 10
      ENDIF
C END Newton;
      RETURN
      END
```

Figure 7: Fortran version of `Newton`

Using the Unix C-shell built-in `time` command, our benchmark comparison revealed that mutation analysis performed by direct execution of the meta-mutant was 4.1 times faster than interpretive execution. This is a strong indication that `MSG` can significantly increase the performance of mutation testing; we expect even more dramatic improvement with future systems.

5 Related Work

Because of the large number of mutant programs that must be generated and run, early designers of mutation analysis systems considered individually creating, compiling, linking, and running each mutant more difficult, and slower, than using an interpretive system [6, 8]. It was considered likely that the cost of compiling large numbers of mutants would be prohibitive. Of the interpreter-based systems that have been developed, `Mothra` is the most recent and comprehensive [11, 15].

In these conventional, interpreter-based mutation analysis systems, the source code is translated into an internal form suitable for interpretive execution and mutation. For each mutant, a mutant generator program produces a "patch" that, when applied to the internal form, creates the desired alternate program. The translated program plus the collection of patches represents a program neighborhood. To run a mutant against a test case, the interpreter dynamically applies the appropriate patch and interpretively executes the resulting alternate internal form program.

A number of attempts to overcome the performance problem have been made. Some approaches attempt to limit the number of mutants that must be run. In *selective mutation* [21], only a subset of the possible mutagens are used, resulting in fewer mutants being created. Preliminary results suggest that selective mutation may provide almost the same test coverage as non-selective mutation under certain conditions. Running only a sample of the

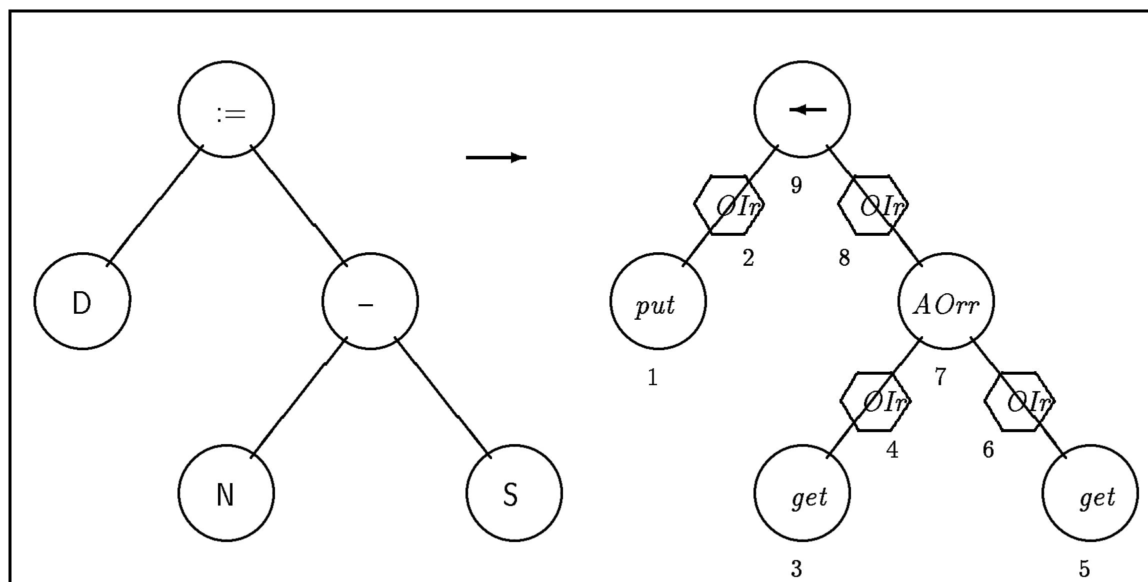

Figure 6: Transforming the Abstract Syntax Tree

mutants [22] has also been suggested. Research into determining what is a statistically appropriate sample size continues. In extreme cases, however, it is necessary to run almost all the mutants.

In other approaches, the use of non-standard computer architectures has been explored. Unfortunately, full utilization of these high performance computers requires an awareness of their special requirements as well as adaptation of software. Work has been done to adapt mutation systems to vector processors [17], to SIMD [16] and hypercube (MIMD) machines [10, 20]. However, it is the very fact that these architectures *are* non-standard that limits the appeal of these approaches. Not only are they not available in most development environments, but testing software designed for one operational environment (machine, operating system, compiler, etc.) on another is fraught with risks.

The approaches above do not squarely address the primary factor that causes conventional systems to be slow: *interpretative execution.* Yet as noted previously, the apparent overhead of compiling many mutant programs outweighs the benefit of increased execution speed. *Compiler-integrated* [12] program mutation seeks to avoid excessive compilation overhead and yet retain the benefit of compiled-speed execution. In this method, the program under test is compiled by a special compiler. As the compilation process proceeds, the effects of mutations are noted and *code patches* that represent these mutations are prepared. Execution of a particular mutant requires only that the appropriate code patch be applied prior to execution. Patching is inexpensive and the mutant executes at compiled-speeds.

Unfortunately, crafting the needed special compiler is an expensive undertaking. Modifying an existing compiler reduces this burden somewhat, but the task is still technically demanding. Moreover, for each new computer and operating system environment, this task must be repeated.

6 Conclusions

In this paper we have presented a novel way to perform mutation testing using program schemata. Mutation analysis systems based on the `MSG` method will exhibit several advantages over interpretive systems. The most obvious advantage is that `MSG` mutation systems are faster than interpreter-based systems. `MSG` systems allow mutants to be executed at compiled speeds, but without having to recompile or store each mutant separately. Although a large number of metaprocedure function calls must be processed, the mutants run as compiled programs and thus execute at machine language speeds.

In addition to improvements in execution speed, `MSG` systems are significantly cheaper to build than interpretive systems. Much of the implementation difficulty of a mutation system is in designing an intermediate language, building a parser, and building an interpreter. These problems are exacerbated in languages that have dynamic memory features, user-defined types, and complicated control features—indeed, much of the reason that `Mothra`

```
        PROCEDURE Newton ( Number:REAL; VAR Sqrt:REAL );
        (*  Find a square root using Newton's method.         *)
        VAR
            NewGuess, Delta, Epsilon : REAL;    (* don't care *)
            dr : REAL;  (* Dummy Real -- used for PUTr target *)
        BEGIN
(* Initialize local variables to default value of 0 *)
NewGuess := 0.0;
Delta    := 0.0;
Epsilon  := 0.0;
(*  0      >Epsilon :=  0.001; *)
            IF SAN(67) THEN
              dr := PUTr(CRr(GETr(40),34),59);
            END;
(*  1      >NewGuess :=  (Number / 2.0) + 1.0; *)
            IF SAN(68) THEN
              dr := PUTr(OIr(AOrr(OIr(AOrr(OIr(GETr(41), 11),
              CRr(GETr(42),35),1),12), CRr(GETr(43),36),2),13),60);
            END;
(*  2      >Sqrt :=  0.0; *)
            IF SAN(69) THEN
              dr := PUTr(CRr(GETr(44),37),61);
            END;
(*  3      >Delta :=  NewGuess - Sqrt; *)
            IF SAN(70) THEN
              dr := PUTr(OIr(AOrr(OIr(GETr(45),14),OIr(GETr(46),15),3),16),62);
            END;
(*  4      >WHILE Delta > Epsilon DO *)
            WHILE LAN(OIl(ROrr(OIr(GETr(47),17), OIr(GETr(48),18),8),32),10) DO
(*  5          >Sqrt :=  NewGuess; *)
                IF SAN(71) THEN
                  dr := PUTr(OIr(GETr(49),19),63);
                END;
(*  6          >NewGuess :=  (Sqrt + (Number / Sqrt)) / 2.0; *)
                IF SAN(72) THEN
                  dr := PUTr(OIr(AOrr(OIr(AOrr(OIr(GETr(50),20),
                  OIr(AOrr(OIr(GETr(51),21), OIr(GETr(52),22),4),23),5),24),
                  CRr(GETr(53),38),6),25),64);
                END;
(*  7          >Delta :=  NewGuess - Sqrt; *)
                IF SAN(73) THEN
                  dr :=
                  PUTr(OIr(AOrr(OIr(GETr(54),26),OIr(GETr(55),27),7),28),65);
                END;
(* EWS *)       IF SAN(75) THEN
(*  8          >IF Delta < 0.0 THEN *)
                IF OIl(ROrr(OIr(GETr(56),29), CRr(GETr(57),39),9),33) THEN
(*  9              >Delta :=  -Delta; *)
                    IF SAN(74) THEN
                      dr := PUTr(OIr(-OIr(GETr(58),30),31),66);
                    END;
(* 11 *)        END;
(* EWS *)       END;
(* 12 *)    END; (* END WHILE *)
(* EWS *)   IF NOT SAN(75) THEN
(* EWS  8*) IF Delta < 0.0 THEN
(* EWS  9*)     Delta :=  -Delta;
(* EWS 11*) END;
(* EWS *)   END;
        END Newton;
```

Figure 8: Newton metamutant (abridged)

(Minor initialization code and the GETr and PUTr metaprocedure routines have been omitted. Original statements are included as comments.)

tests Fortran programs is because Fortran-77 does not have these features. Since the MSG method leaves the problems of providing run-time semantics and environment to pre-existing compilers and run-time libraries, MSG systems are smaller and easier to build, allowing us to quickly develop mutation tools for a variety of languages.

MSG systems also provide more realistic testing. Since the MSG method produces a compilable program in the same language as the program being tested, testing can take place using the same compiler and environment that will be used by the program under test. Hence the program is tested in the same operational environment that it will be used in, and retains all or most of its original operational behavior.

An important advantage of MSG mutation systems is their innate portabilty. Because MSG systems operate at the source-level, they can easily be moved from machine to machine, or compiler to compiler. For example, a network computer system with different architectures (for example, Sun3s and Sun4s) could be utilized simply by recompiling the MSG system for each different type of machine. This architecture independence also makes heterogeneous distributed computing implementations easier.

Lastly, since the ability to compile and run a program P is provided by an existing L-language compiler, it is possible for an MSG system to be incomplete and yet provide partial functionality. Although the driver must be substantially complete, not all the metaprocedures need to be written nor all the metamutation transformation mechanisms defined. Unimplemented mutagens result in P source text passing through to the compiler unaltered. This is in contrast to an interpreter-based system where virtually the entire translator and run-time interpreter must be finished for programs to be executed and tested. This characteristic of schema-based systems encourages incremental implementations and also allows greater freedom to experiment with the mutagenic operators.

Although much faster than interpretive systems, MSG systems may be somewhat slower than *compiler-integrated* methods. *Compiler-integrated* systems are not burdened by metaprocedure function call overhead, so the execution speed of a mutant is identical with that of a single program that exhibits the fault of the mutation. We feel that the portability and ease of constructing an MSG system far outweighs this execution speed difference. Unless mutation systems using object code patches are built into compilers from the start, modifying pre-existing compilers is unlikely to be practical.

It is interesting to note that the MSG method is orthogonal to many of the approaches discussed in section 5. For example, schema-based mutation could be performed in concert with a *compiler-integrated* method. Similarly, the mutant sampling strategy could be used regardless of the underlying mutation analysis mechanism, and an MSG system could implement weak mutation. In addition, there is no reason to believe that MSG systems could not be successfully adapted to run in a distributed computing environment.

References

[1] Allen Troy Acree, Jr. *On Mutation.* Ph.D. dissertation, Georgia Institute of Technology, August 1980.

[2] Hiralal Agrawal, Richard A. DeMillo, R. Hathaway, William Hsu, Wynne Hsu, Edward W. Krauser, Rhonda J. Martin, Aditya P. Mathur, and Eugene H. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, March 20 1989.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[4] D. Baldwin and Frederick G. Sayward. Heuristics for Determining Equivalence of Program Mutations. Research Report 276, Department of Computer Science, Yale University, New Haven, CT, 1979.

[5] Orit Baruch and Shmuel Katz. Partially Interpreted Schemas for CSP Programming. *Science of Computer Programming*, 10(1):1–18, February 1988.

[6] Timothy A. Budd. *Private correspondence*, February 24 1992.

[7] Timothy A. Budd and Dana Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1):31–45, November 1982.

[8] Timothy A. Budd, Richard J. Lipton, Frederick G. Sayward, and Richard A. DeMillo. The Design of a Prototype Mutation System for Program Testing. In *Proceedings of the National Computer Conference*, pages 623–627,

Anaheim, CA, June 5–8 1978. The Association for Computing Machinery, AFIPS Press, Montvale, NJ. Vol. 47.

[9] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. Ph.D. dissertation, Yale University, May 1980.

[10] ByoungJu Choi and Aditya P. Mathur. High-Performance Mutation Testing. *The Journal of Systems and Software*, 20(2):135–152, February 1993.

[11] Richard A. DeMillo, Dany S. Guindi, Kim N. King, W. Michael McCracken, and A. Jefferson Offutt. An Extended Overview of the Mothra Software Testing Environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Alberta, Canada, July 19–21 1988. IEEE Computer Society Press.

[12] Richard A. DeMillo, Edward W. Krauser, and Aditya P. Mathur. Compiler-Integrated Program Mutation. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference (COMPSAC)*, pages 351–356, Tokyo, Japan, September 11–13 1991. IEEE Computer Society Press.

[13] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[14] Richard G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[15] Kim N. King and A. Jefferson Offutt. A Fortran Language System for Mutation-based Software Testing. *Software—Practice and Experience*, 21(7):685–718, July 1991.

[16] Edward W. Krauser, Aditya P. Mathur, and Vernon J. Rego. High Performance Software Testing on SIMD Machines. *IEEE Transactions on Software Engineering*, SE-17(5):403–423, May 1991.

[17] Aditya P. Mathur and Edward W. Krauser. Mutant Unification for Improved Vectorization. Technical Report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, April 25 1988.

[18] A. Jefferson Offutt and William Michael Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. Technical Report 92-102, Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA, November 1992.

[19] A. Jefferson Offutt and Stephen D. Lee. Instructive Mutation System from Clemson University: System Documentation. Technical Report 91-121, Computer Science Department, Clemson University, Clemson, SC, 1991.

[20] A. Jefferson Offutt, Roy P. Pargas, Scott V. Fichter, and Prashant K. Khambekar. Mutation Testing of Software Using a MIMD Computer. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages II–257–266, St. Charles, IL, August 17–21 1992.

[21] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An Experimental Evaluation of Selective Mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, Baltimore, MD, May 17–21 1993. IEEE Computer Society Press. *To appear.*

[22] Mehmet Şahinoğlu and Eugene H. Spafford. A Sequential Statistical Procedure in Mutation-Based Testing. In *Proceedings of the 28th Annual Spring Reliability Seminar*, pages 127–148, Boston, MA, April 19 1990. Central New England Council of IEEE.

[23] Roland H. Untch. Mutation-based Software Testing Using Program Schemata. In *Proceedings of the 30th Annual ACM Southeast Conference*, pages 285–291, Raleigh, NC, April 8–10 1992. The Association for Computing Machinery.

[24] Elaine J. Weyuker. On Testing Non-testable Programs. *The Computer Journal*, 25(4):465–470, November 1982.

[25] D. Wu, M. A. Hennell, D. Hedley, and I. J. Riddell. A Practical Method for Software Quality Control via Program Mutation. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 159–170, Banff, Alberta, Canada, July 19–21 1988. IEEE Computer Society Press.