Technical Report 95–115
Department of Computer Science
Clemson University

SCHEMA-BASED MUTATION ANALYSIS: A NEW TEST DATA ADEQUACY ASSESSMENT METHOD

A Dissertation

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

Computer Science

---

by

Roland H. Untch

December 1995

Committee: Mary Jean Harrold (Dissertation Advisor)
A. Jefferson Offutt
Harold C. Grossman
Brian A. Malloy
Ronald H. Nowaczyk

DEDICATION

This work is dedicated to my first and most important teachers,

my parents,

Michael and Mathilde.

I love you both.

ACKNOWLEDGEMENTS

I am deeply indebted to Mary Jean Harrold for her guidance and patience. I am grateful to Jeff Offutt for many things, including introducing me to the world of software testing and computer science research. I thank Harold Grossman for his candor and encouragement over the years, both were greatly appreciated. I thank Brian Malloy and Ron Nowaczyk for serving on my Ph.D. committee. My fellow graduate students Jim O'Connor, Darrell Suggs, and Gregg Rothermel deserve my acknowledgement and my gratitude—I would not have succeeded without their help. Finally, I would like to thank Sandra, my wife, for her endless support and encouragement.

ABSTRACT

Mutation-based software testing, or *mutation testing*, is a powerful testing technique applied primarily at the unit software level. Central to mutation testing is the need to analyze a test set to determine a quality measure called the *mutation adequacy score*; this assessment process is called *mutation analysis*. Unfortunately, the conventional method of performing mutation analysis, which requires interpreting many slightly different versions of the same program, has significant problems. Automated mutation analysis systems based on the conventional interpretive method are slow, laborious to build, and usually unable to completely emulate the intended operational environment of the software being tested.

This research presents a solution to these problems: the Mutant Schema Generation (MSG) method. Rather than mutating an intermediate form of the program that then must be interpreted, this new method describes how to encode all mutations into one source-level program, a "metamutant". 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. An approach to automatically generating metamutants using attribute grammars is also presented.

An MSG-based prototype mutation analysis system, `TUMS`, was designed and implemented to demonstrate the automated generation of metamutants and to allow empirical performance studies to be conducted. Benchmarks show `TUMS` significantly faster than `Mothra`, a conventional interpretive mutation analysis system, with speed-ups as high as an order-of-magnitude observed. Additional studies are reported that contrast the performance of `TUMS` to a hypothetical "ideal" mutation analysis system.

We conclude that high performance mutation analysis is possible through the creation and instantiation of mutant schemata and that the MSG method described in this dissertation is a viable and desirable approach for building automated mutation analysis systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Programs are tested by executing them against test inputs and examining the resulting outputs for errors. The intent of testing 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[1]. Using a criterion, the testing of a program ceases when either the set of test cases meets the minimum quality goal imposed by the criterion or an incorrect output is noted. This process is illustrated in Figure 1. For example, the *branch coverage* criterion states that the set of test cases must cause each branch point in a program to be traversed at least once [1]. *Data flow testing* criteria require the set of test cases to exercise certain subpaths from a point in the program where a variable is given a value (is *defined*) to points where that variable definition is subsequently *used* [2]. Changing the required combinations of definitions and uses yields a variety of data flow testing techniques [3, 4, 5, 6, 7].

Mutation-based software testing, or *mutation testing*, is a powerful testing technique that uses an adequacy criterion [8, 9, 10, 11, 12, 13]. 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

---

[1] In testing literature, the word *program* is typically used to denote the software under test. This may be a complete program or some smaller unit, such as a procedure.

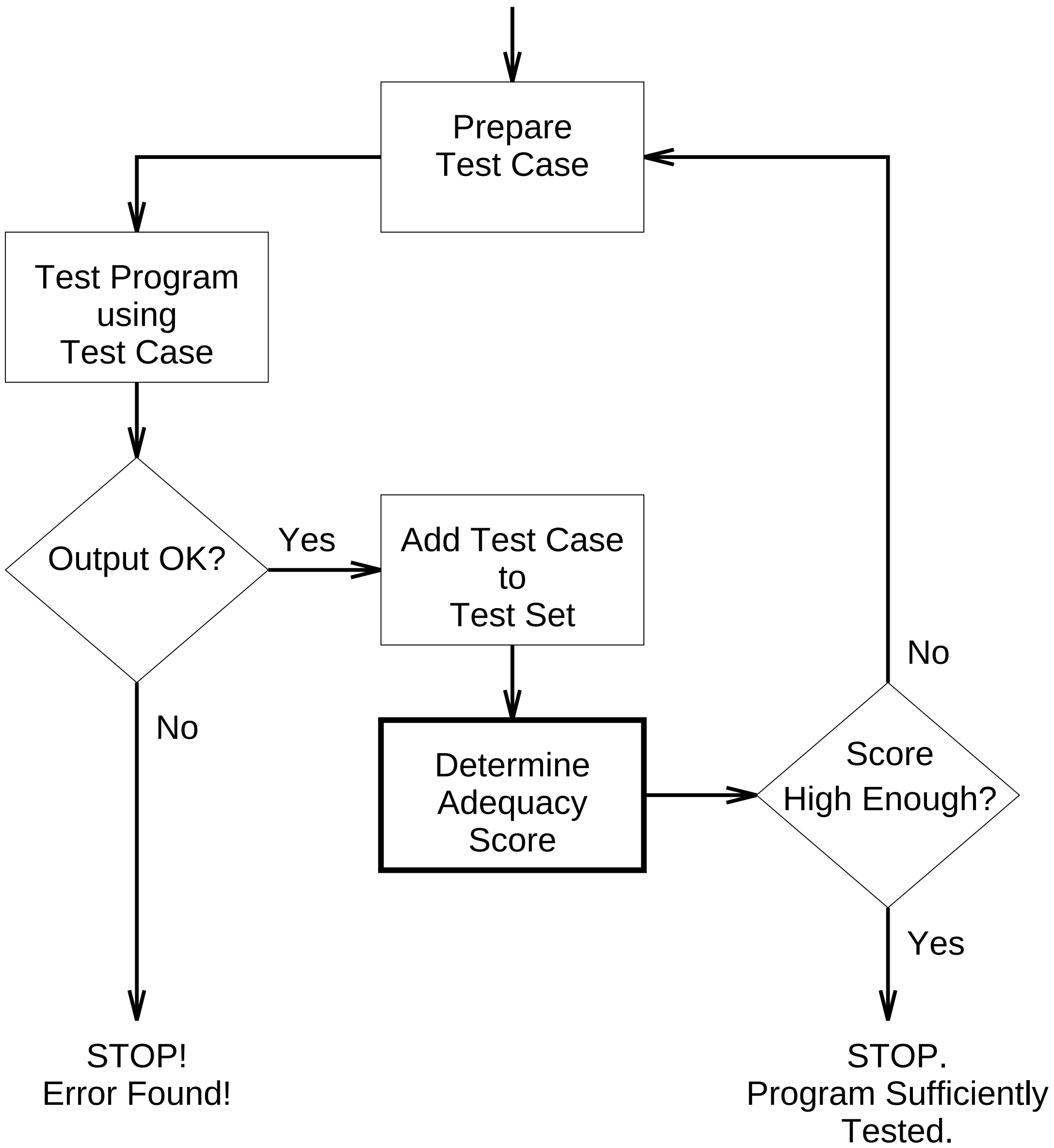

Figure 1. Adequacy-based testing process.

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.

This dissertation presents a new method of performing mutation analysis that solves these problems. Rather than mutating an intermediate form of the program that then must be interpreted, the Mutant Schema Generation (MSG) method describes how 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.

Mutation Analysis

Mutation analysis is a *white-box* testing technique. It shares with other white-box testing techniques the premise that, since we are seeking to increase our confidence in the probable correctness of a particular program, good test data must be tied very closely to the form and structure of this program [14]. Thus the quality of test data must be measured relative to the program being tested. A good test set for one program may not be good for another, even if the two programs are identical in function [15].

Mutation analysis asserts 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. This assertion is supported by a number of theoretical and empirical studies [10, 11, 16, 17, 18, 19, 20]. The assertion is also intuitively compelling. Consider a test set where the original program and the set of alternate programs produce the same output results. On the basis of this testing we have no more reason to believe that the original program is correct than any of the alternatives, hence that test set is of low utility and quality.

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 so-called "Oracle" problem [21]. This problem 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 are produced. Each alternate program, P_i, known as a *mutant* of P, is formed by modifying a single element of P according to some predefined modification rule.[2] Such modification rules, G, are called *mutagenic operators* or *mutagens*[3]. 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 [24]. Figure 2 illustrates the relationship of these items.

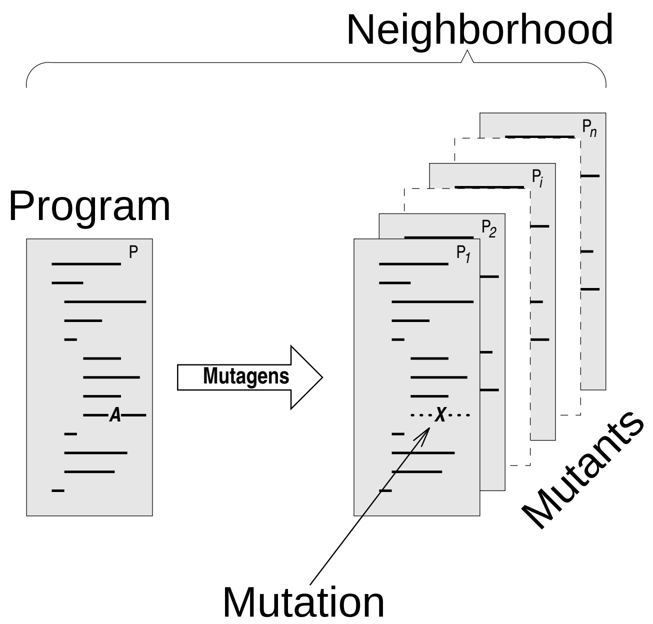

Figure 2. Components of a program neighborhood.

[2]Only a *single* syntactic change is needed because of the *coupling effect* [22].

[3]The terminology varies; they are also sometimes called *mutant operators*, *mutation operators*, *mutation transformations*, and *mutation rules* [23]. Acree [10] uses the term *mutagenic operator*; in biology, a mutagenic substance or factor is simply called a *mutagen*.

Each mutant is run against the test cases in T. If a mutant produces a result different than that of the original program on some test case in T, 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 [25, 26, 27], 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 specific measure used to express test set quality is the *mutation adequacy score*, or *MS*, which 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 the literature, 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 proportional to the product of the number of data references and the number of data objects [11, 28], which is typically a large number. For example, 707 mutants get generated for the 23 line `SUMSQRT` program shown in Figure 3.

Because of the large number of mutant programs that must be generated and run, designers of early mutation analysis systems considered individually creating, compiling, linking, and running each mutant more difficult, and slower, than using an interpretive system.[4] Recent research confirms that such a *separate compilation* approach would likely be plagued by a significant *compilation bottleneck* [29]; the cost of compiling large numbers of mutants would likely be prohibitive. Several systems based on interpretation were developed [30, 31, 32]. `Mothra` is the most recent and comprehensive of such interpreter-based systems [33, 34].

```
void SUMSQRT( float N, float *SUM )
{
    float NUMBER, SQRT, GUESS, DELTA, EPS;
    EPS = 0.001;
    *SUM = 0.0;
    NUMBER = 1.0;
    while (NUMBER <= N)
    {
        GUESS = NUMBER / 2.0 + 1.0;
        SQRT = 0.0;
        DELTA = GUESS - SQRT;
        while (DELTA > EPS)
        {
            SQRT = GUESS;
            GUESS = (SQRT + NUMBER / SQRT) / 2.0;
            DELTA = GUESS - SQRT;
            if (DELTA < 0.0)
                DELTA =  -DELTA;
        }
        *SUM = *SUM + SQRT;
        NUMBER = NUMBER + 1.0;
    }
}
```

Figure 3. `SUMSQRT` sum of square roots program.

In these interpreter-based mutation analysis systems, the source code is translated into an internal form suitable for interpretive execution and mutation [10]. For each mutant, a mutant generator program produces a "patch" that, when applied to the internal form, creates the desired alternate program. Such a patch is called a *mutant descriptor*. The translated program plus the collection of mutant descriptors represents a program neighborhood. To run a mutant against a test case, the interpreter dynamically applies the appropriate mutant descriptor patch and interpretively executes the resulting alternate internal form program.

[4] Timothy A. Budd. *Private Correspondence*, February 24 1992.

Although providing a compact representation of the program neighborhood and yielding workable systems, such conventional mutation analysis systems exhibit the performance characteristics typical of interpretive systems: *they are slow.* As one study noted, "current implementations of mutation tools are unacceptably slow and are only suitable for testing relatively small programs" [35]. 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 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.

Related Work

A number of attempts to overcome the performance problem have been made.

Reduced Neighborhoods

In several approaches, execution cost is lowered by running only a subset of the mutants. We shall use the term *reduced neighborhood* mutation to refer to any approach that uses a reduced set of mutants; we shall use the term *standard neighborhood* mutation to refer to approaches that use a standard set of mutants.

Acree [10] and Budd [11] proposed a form of reduced neighborhood mutation in which a random sample of 10% of the mutants in a program neighborhood is used. The effects of varying the sampling percentage from 10% to 40% in steps of 5% were later investigated by Wong [20]. Although sampling mutants does lower execution cost, it unfortunately also weakens the mutation adequacy criterion. Wong observed that mutation testing using the weakened criterion was less effective at fault detection than using the full criterion. A 10% sample, for example, was found to be 16% less effective than standard neighborhood mutation in its fault detection effectiveness.

Ṣahinoğlu and Spafford propose another sampling approach that does not use samples of some *a priori* fixed size [36]. In this approach, based on a Bayesian sequential probability ratio test, mutants are randomly selected from the program neighborhood until sufficient evidence has been collected to determine that a statistically appropriate sample size has been reached. However this approach requires that the costs of faulty acceptance, rejection, and sampling be ascertained. Determining these parameters is difficult and subject to guesswork. Moreover, in the worst case, it is necessary to run almost all the mutants [37].

Another way of obtaining a reduced program neighborhood is to restrict the set of mutagens used in generating the neighborhood. However it is not obvious what reduced set of mutagens should be used. In *constrained mutation*, as proposed by Mathur [38], those mutagens thought to be most significant in fault detection are selected. In Wong's investigation of constrained mutation using the `Mothra` system [20], only two mutagens (`ABS` and `ROR`) are used. In *N-selective mutation*, as proposed by Offutt et al. [39], mutagens that produce a large number of mutants are excluded. Later work in selective mutation divided mutagens into categories based on the syntactic elements they affect [28]. Different reduced neighborhoods are produced by restricting the mutagens used by category. For both constrained and selective mutation, research into the effects the reduced program neighborhood has on the mutation adequacy criterion continues.

Parallel Execution

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 and adaptation of software. Work has been done to adapt mutation analysis systems to vector processors [40], SIMD machines [35], Hypercube

(MIMD) machines [41, 42], and Network (MIMD) computers [43]. 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 [29].

Improved Interpretation Techniques

Weiss and Fleyshgakker describe algorithms that improve the run time complexity of conventional mutation analysis systems at the expense of increased space complexity [44, 45]. In the best case, these techniques can improve the speed by a factor proportional to the average number of mutants per program statement. In the worst case, there is no improvement.

Machine Code Patching

The approaches above do not squarely address the primary reason that conventional systems are slow: interpretive execution. DeMillo, Krauser, and Mathur developed a *compiler-integrated* program mutation scheme that avoids much of the overhead of the compilation bottleneck and yet is able to execute compiled code [46, 47]. This method mimics conventional methods but works with (compiler-generated) object code instead of some interpreter internal form. 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.

Scope and Goal of This Research

The goal of this research is to reduce the duration of time needed to perform standard neighborhood mutation analysis. Specifically, we are interested in reducing the time needed to execute mutants without concomitantly introducing excessive mutant generation, compilation, or storage costs.

We present a new method of performing mutation analysis that introduces a construct we call a mutant schema, describes how mutant schemata can be devised to model program neighborhoods, and details a technique for automatically generating mutant schemata. The overall thesis of this research is that *high performance mutation analysis is possible through the creation and instantiation of mutant schemata.*

To answer the question of how fast schema-based mutation analysis is, a working prototype mutation analysis system based on the precepts of the new method was constructed. Using this prototype system, we empirically established large reductions in the duration of time needed to perform mutation analysis.

Organization of This Dissertation

Chapter II describes the new method for performing mutation analysis. We first present the overall strategy behind the new method. We next explore some of the details that must be addressed when applying the method. We conclude by describing an attribute grammar-based technique for automatically generating mutant schemata. In Chapter III we introduce the `TUMS` prototype mutation analysis system. The goal and design of the system are presented followed by highlights of some of the implementation details. We also discuss using the prototype system. Chapter IV presents several empirical results that relate the performance of mutation analysis using the new method to previous and hypothesized methods. Chapter V summarizes our work and discusses the advantages and disadvantages of the new method. The significance of this work is assessed and several interesting suggestions for future work are presented.

CHAPTER II

THE MSG METHOD

This chapter presents our Mutant Schema Generation (`MSG`) method for performing mutation analysis. This chapter introduces a construct we call a mutant schema, describes how mutant schemata can be devised to model program neighborhoods, and outlines our technique for automatically generating mutant schemata.

Mutation Analysis Using Mutant Schemata

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 [48], 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* [49, 50]. 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. These are described below.

The essence of the `MSG` method of mutation analysis lies in the creation of a specially parameterized program called a *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. Using the `SUMSQRT` program from Chapter I for illustration (see Figure 3), recall that each mutant of a program P is formed as a result of a single modification to some statement in P. Thus each mutant of SUMSQRT differs from the original by only one mutated statement (and only one change within the 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 by Agrawal et al. for the C language [51]; these rules are typical of those in current use [34, 52].

Consider the *arithmetic operator replacement* (OAAN) mutagen, which states that each occurrence of an arithmetic operator is replaced by each of the other possible arithmetic operators. Applying this mutagen to the assignment statement of line 11 of SUMSQRT

```
DELTA = GUESS - SQRT;
```

yields these four mutations.

```
DELTA = GUESS + SQRT;
DELTA = GUESS * SQRT;
DELTA = GUESS / SQRT;
DELTA = GUESS % SQRT;
```

In the MSG method, we represent these mutations "generically" as

DELTA = GUESS *ArithOp* SQRT;

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

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

```
DELTA = AO_(GUESS,SQRT,15);
```

where the AO_ function performs one arithmetic operation. The third argument, "15" in this example, is used to identify the original, or default, operation that is performed in the absence of a mutation—in this case a subtraction. "AO_" is an example of a *metaprocedure*, a function that corresponds to an abstract entity in the schema. We say a statement that has been changed to reflect such a generic form has been *metamutated*. A *metamutation* is a syntactically valid change that embodies other changes.

A *change point* is a location in the program where a mutation can occur and corresponds to a place where an abstract entity can be inserted or substituted for an actual syntactic item. An *implicit* change point is a location just before an expression and marks an abstract entity insertion point. An *explicit* change point is either an entire statement or a token representing an operator or operand. Each item at an explicit change point is subject to substitution by an abstract entity. Within the assignment statement of line 11 of SUMSQRT, there are five explicit change points located at the two operator and three operand tokens.

To further illustrate the metamutation of operators, we use the *binary operator replacement* (Obor) mutagen, which states that each occurrence of a binary operator is replaced by each of the other legal binary operators.[5] In the original statement above there are two binary operators—the assignment operator, "=", and the minus sign, "-". Applying the Obor mutagen to line 11 yields these 27 mutations.

```
DELTA |=  GUESS -  SQRT;
DELTA ^=  GUESS -  SQRT;
DELTA &=  GUESS -  SQRT;
DELTA <<= GUESS -  SQRT;
DELTA >>= GUESS -  SQRT;
DELTA +=  GUESS -  SQRT;
DELTA -=  GUESS -  SQRT;
DELTA *=  GUESS -  SQRT;
DELTA /=  GUESS -  SQRT;
DELTA %=  GUESS -  SQRT;
DELTA =   GUESS || SQRT;
DELTA =   GUESS && SQRT;
DELTA =   GUESS |  SQRT;
DELTA =   GUESS ^  SQRT;
DELTA =   GUESS %  SQRT;
DELTA =   GUESS == SQRT;
DELTA =   GUESS != SQRT;
DELTA =   GUESS <  SQRT;
DELTA =   GUESS >  SQRT;
DELTA =   GUESS <= SQRT;
DELTA =   GUESS >= SQRT;
DELTA =   GUESS << SQRT;
DELTA =   GUESS >> SQRT;
DELTA =   GUESS +  SQRT;
DELTA =   GUESS *  SQRT;
DELTA =   GUESS /  SQRT;
DELTA =   GUESS %  SQRT;
```

From these, we obtain the generic representation

DELTA *BinaryOp* GUESS *BinaryOp* SQRT;

---

[5]Note that the mutations induced by the OAAN (arithmetic operator replacement) mutagen are a subset of those induced by the Obor (binary operator replacement) mutagen.

which in turn can be recast as

```
BO_(DELTA,BO_(GUESS,SQRT,15,4),37,1);
```

where each invocation of the `BO_` metaoperator function performs one binary operation. The third argument of the `BO_` functions, "`15`" and "`37`" in this example, once again identifies the default binary operations to be performed: subtraction and assignment, respectively. In addition, we introduce a fourth argument to identify the change point, or location, in the program where this function is invoked. Without these change point arguments, "`4`" and "`1`" in this example, there would be no way of distinguishing the two invocations of the `BO_` function.

All mutations produced from applying standard mutagens can be represented by metamutations. Note that in applying the standard mutagens, more than just the explicit operators in an expression get mutated. Operands get mutated and unary operators get inserted at implicit change points. Figure 4 shows all the mutations of line 11 that result from applying the complete set of C mutagens. The following statement

```
BO_(REF_(L_,8,2),OI_(BO_(OI_(REF_(L_,7,6),5),OI_(REF_(L_,6,8),7),15,4),3),37,1);
```

embodies all of these alternatives.

In the `MSG` method, we produce the metamutant of P by metamutating each of the statements of P in a manner similar to that just illustrated. For reference, the metamutant of `SUMSQRT` that represents the functionality of the 707 mutants resulting from applying the complete set of `TUMS` mutagens[6] is listed in Appendix A.

While generating the metamutant of P, a list of *mutant descriptors*, D, is produced. These mutant descriptors are used to dynamically instantiate the metamutant to function as some mutant of P. There is one mutant descriptor for each mutant in the neighborhood of P. Each mutant descriptor is a set of metamutant parameter values that "describe" a particular mutation. A mutant descriptor contains, at minimum, two items: a change point and an alternative action to be taken at that change point. A "driver" or "harness"

---

[6] The set of `TUMS` mutagens is slightly different than the set of C mutagens defined by Agrawal et al. [51] and are described in Chapter III.

```
11  Original=>  DELTA = GUESS - SQRT;
    ---------   ----------------------------------
    [Mutagens] / [Mutations]
    -----       ----------------------------------
    [VLSR]      N = GUESS - SQRT;
    [VLSR]      NUMBER = GUESS - SQRT;
    [VLSR]      SQRT = GUESS - SQRT;
    [VLSR]      GUESS = GUESS - SQRT;
    [VLSR]      EPS = GUESS - SQRT;
    [VLSR]      *SUM = GUESS - SQRT;
    [OEBA]      DELTA |= GUESS - SQRT;
    [OEBA]      DELTA ^= GUESS - SQRT;
    [OEBA]      DELTA &= GUESS - SQRT;
    [OESA]      DELTA <<= GUESS - SQRT;
    [OESA]      DELTA >>= GUESS - SQRT;
    [OEAA]      DELTA += GUESS - SQRT;
    [OEAA]      DELTA -= GUESS - SQRT;
    [OEAA]      DELTA *= GUESS - SQRT;
    [OEAA]      DELTA /= GUESS - SQRT;
    [OEAA]      DELTA %= GUESS - SQRT;
    [VDTR]      DELTA = TOZ_(GUESS - SQRT);
    [VDTR]      DELTA = TOP_(GUESS - SQRT);
    [VDTR]      DELTA = TON_(GUESS - SQRT);
    [VTWD]      DELTA = -(GUESS - SQRT);
    [VTWD]      DELTA = SUCC_(GUESS - SQRT);
    [VTWD]      DELTA = PRED_(GUESS - SQRT);
    [VDTR]      DELTA = TOZ_(GUESS) - SQRT;
    [VDTR]      DELTA = TOP_(GUESS) - SQRT;
    [VDTR]      DELTA = TON_(GUESS) - SQRT;
    [VTWD]      DELTA = -GUESS - SQRT;
    [VTWD]      DELTA = SUCC_(GUESS) - SQRT;
    [VTWD]      DELTA = PRED_(GUESS) - SQRT;
    [VLSR]      DELTA = N - SQRT;
    [VLSR]      DELTA = NUMBER - SQRT;
    [VLSR]      DELTA = SQRT - SQRT;
    [VLSR]      DELTA = DELTA - SQRT;
    [VLSR]      DELTA = EPS - SQRT;
    [VLSR]      DELTA = *SUM - SQRT;
    [VLCR]      DELTA = 0.001 - SQRT;
    [VLCR]      DELTA = 0.0 - SQRT;
    [VLCR]      DELTA = 1.0 - SQRT;
    [VLCR]      DELTA = 2.0 - SQRT;
    [OALN]      DELTA = GUESS || SQRT;
    [OALN]      DELTA = GUESS && SQRT;
    [OABN]      DELTA = GUESS | SQRT;
    [OABN]      DELTA = GUESS ^ SQRT;
    [OAAN]      DELTA = GUESS & SQRT;
    [OABN]      DELTA = GUESS == SQRT;
    [OARN]      DELTA = GUESS != SQRT;
    [OARN]      DELTA = GUESS < SQRT;
    [OARN]      DELTA = GUESS > SQRT;
    [OARN]      DELTA = GUESS <= SQRT;
    [OARN]      DELTA = GUESS >= SQRT;
    [OARN]      DELTA = GUESS << SQRT;
    [OASN]      DELTA = GUESS >> SQRT;
    [OASN]      DELTA = GUESS + SQRT;
    [OAAN]      DELTA = GUESS * SQRT;
    [OAAN]      DELTA = GUESS / SQRT;
    [OAAN]      DELTA = GUESS % SQRT;
    [VDTR]      DELTA = GUESS - TOZ_(SQRT);
    [VDTR]      DELTA = GUESS - TOP_(SQRT);
    [VDTR]      DELTA = GUESS - TON_(SQRT);
    [VTWD]      DELTA = GUESS - -SQRT;
    [VTWD]      DELTA = GUESS - SUCC_(SQRT);
    [VTWD]      DELTA = GUESS - PRED_(SQRT);
    [VLSR]      DELTA = GUESS - N;
    [VLSR]      DELTA = GUESS - NUMBER;
    [VLSR]      DELTA = GUESS - GUESS;
    [VLSR]      DELTA = GUESS - DELTA;
    [VLSR]      DELTA = GUESS - EPS;
    [VLSR]      DELTA = GUESS - *SUM;
    [VLCR]      DELTA = GUESS - 0.001;
    [VLCR]      DELTA = GUESS - 0.0;
    [VLCR]      DELTA = GUESS - 1.0;
    [VLCR]      DELTA = GUESS - 2.0;
```

Figure 4. `SUMSQRT` line 11 and its 71 mutations.

invokes the metamutant and directs which mutant is to be instantiated by selecting the corresponding mutant descriptor from D. As the metamutant execution progresses through each change point, a check is made to determine whether the change point matches that in the mutant descriptor. If so, the alternative (mutated) action is taken; otherwise the default (unmutated) action is performed.

In addition to selecting mutant descriptors from D and invoking the metamutant, 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.

A conceptual model of MSG-based mutation analysis is given in Figure 5. 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 abstracting the program neighborhood N. The program neighborhood is obtained by applying the mutagens G to the program P.

$$\left.\begin{matrix} P \\ G \end{matrix}\right\} \overset{\text{mutation}}{\Longrightarrow} N \overset{\text{abstraction}}{\Longrightarrow} \left\{\begin{matrix} M \\ D \end{matrix}\right\} \overset{\text{instantiation}}{\Longrightarrow} P_i \underset{\underset{T}{\uparrow}}{\overset{\text{execution}}{\Longrightarrow}} MS_G(P,T)$$

Figure 5. Model of MSG-based mutation analysis.

Metamutant Design

Metaprocedures are syntactically valid representations of the abstract entities found in mutant schemata and are unique to the MSG method. Metaprocedures are used wherever a choice among alternatives is needed. We categorize metaprocedures 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.

Metaoperand procedures evaluate to a program reference: either a program constant, variable, or scalar reference expression such as an array reference or a dereferenced pointer. The actual program reference 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.

Complications

The discussion so far has provided a glimpse of the high-level strategy underlying metamutation. In that strategy, mutatable elements in a program are replaced by abstract entities that, in turn, are represented by metaprocedures. It is important to note that some metaprocedures must deal with complications that preclude their implementation as functions or subroutines. These complications are: (1) mixed mode expressions, (2) short-circuit evaluation, (3) structural mutations, and (4) instantiation overhead.

The Need for Dynamic Typing

Mixed mode expressions, in languages where variables have a definite unchanging data type (i.e., are *statically typed*), are those in which the operands within the expression are of different data types. Although some older languages (notably old dialects of Fortran) prohibit mixed mode expressions, such expressions are valid in most current programming languages, such as C. Consequently mutations that produce mixed mode expressions from single mode expressions must be accommodated.

Assume a program contains the integer variables `Na` and `Nb` and the floating point variables `Fx` and `Fy`. Let the assignment statement

```
Fx = Na / Nb;
```

be partially metamutated to

```
Fx = BO_(Na,Nb,17,1);
```

then the corresponding BO_ function prototype might be written as

```
int BO_(int Operand1, int Operand2, int DefaultOp, int ChangePoint);
```

since the two operands are integer and (assuming C semantics) the resulting quotient is also an integer, truncated as necessary. However as a result of applying the *scalar for scalar replacement* (Vssr) mutagen, one of the possible mutations is

```
Fx = Fy / Nb;
```

where the first operand of the division is now a float and not an int and where (assuming C semantics) the resulting quotient is no longer truncated to an integer but is real-valued. Clearly if the BO_ metaoperator is declared using the function prototype above, it will be unable to function as this required mixed mode expression.

Only one metaoperand function is needed in languages that permit mixed mode expressions. This single function, which we call REF_, must be able to evaluate to any of the program's references. The same declaration dilemma we encountered in trying to declare BO_ occurs when trying to declare the REF_ function: what return type do we declare a function that must return a multiplicity of data types?

The answer to the declaration dilemma lies in defining a "generic data object" type that can store references to the actual data objects in the program. This generic data type, which we call tREF_, contains a pointer to an actual data object plus information from which we can infer the data type of the data object. Using this generic data type, the BO_ function prototype could be written as

```
tREF_ BO_(tREF_ Operand1, tREF_ Operand2, int DefaultOp, int ChangePoint);
```

The use of the generic data type in a metamutant introduces the need for *dynamic typing*. Each use of a program reference and each evaluation of an operational expression requires a determination at run-time as to the underlying data type(s) of the data object(s) being manipulated. Each metaprocedure must contain logic to support this dynamic typing.

Short-circuit Evaluation

Short-circuit evaluation refers to the evaluation strategy where the second operand of a Boolean operation might, under certain circumstances, not be evaluated. For example in C, given the Boolean expression

```
(D!=0 && N/D>5)
```

the evaluation of the second operand of the Boolean AND operator (“`&&`”) leads to an execution error if the value of `D` is zero; the first operand is included to insure that the divide by zero error does not occur. The intent is that if the left expression evaluates to `FALSE`, the right expression is never evaluated. Since C implements short-circuit evaluation, this intent is satisfied. Assuming a Boolean `AND_` function exists, we partially metamutate our expression to be

```
AND_(D!=0, N/D>5)
```

where the first and second arguments of `AND_` are assumed to be Boolean valued. Unfortunately the arguments to `AND_` are *fully evaluated before the function is even invoked*; the short-circuiting does not occur and an execution error may well ensue. Similar concerns exist with the Boolean OR (“`||`”) operator.

The `AND_` function was used to simplify the example above. Since the Boolean operators are binary operators, in reality the binary operator `BO_` metaoperator would be used wherever a Boolean AND or OR operator occurred. Unfortunately the need for short-circuit evaluation rules out implementing the `BO_` metaoperator as a function.

To implement the `BO_` metaoperator, a scheme is needed where the right operand is evaluated only as needed. Figure 6 depicts such a scheme. First the left operand is fully evaluated and its value stored. Next it is determined if the current operation is a Boolean operation. If the operation is a Boolean operation, the value of the left operand and the type of Boolean operation requested are used to determine whether the right operand is evaluated. For AND operations, if the left operand value is FALSE the right operand is not evaluated and the metaoperator returns FALSE. For OR operations, if the left operand value is TRUE the right operand is not evaluated and the metaoperator returns TRUE.

In all other cases the right operand is evaluated and a function, which we call BinOp_, is invoked that takes the stored left operand value and the newly evaluated right operand value and performs the desired binary operation. Because of evaluation order, in composite expressions it may be necessary to store the values of several left operands before they are used in an operation. We associate a temporary storage location with each change point and store the value of the left operands in the corresponding storage location. Thus the BinOp_ function draws the value of the left operand from this storage location but gets the value of the right operand via a parameter.

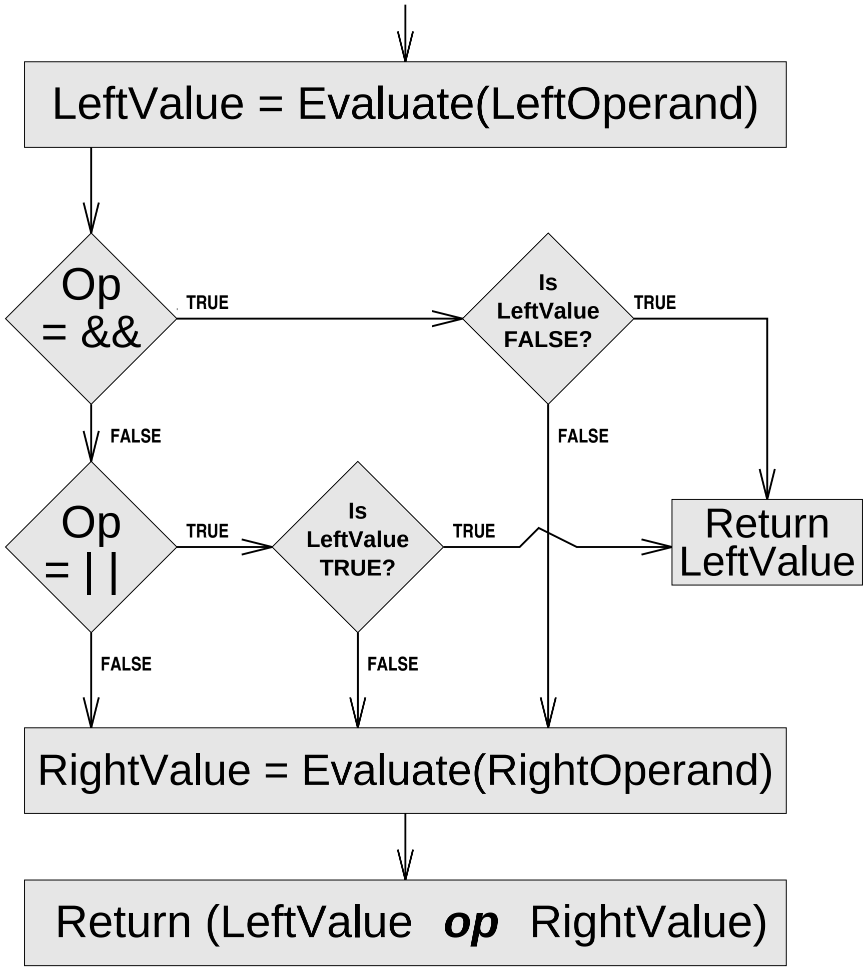

Figure 6. Short-circuit evaluation of a binary operator.

Although the BO_ metaoperator is not a function, it is conceptually convenient to think of it as one and use it as such. In C it is possible to define the BO_ metaoperator as a macro that syntactically resembles a function. The macro definition

```
#define BO_(LEFTARG,RIGHTARG,ORG,CP) \
  ( LA_(LEFTARG,ORG,CP) ? BinOp_(RIGHTARG,ORG,CP) : Left_[CP] )
```

achieves this while implementing the logic shown in Figure 6.

Structural Mutations

The examples of mutation given so far have all dealt with operator or operand substitutions. Some mutagens, however, mutate entire statements and may induce changes to the very structure of the program. For example, the *statement deletion* (SSDL) mutagen, which systematically deletes each statement of the program, causes such structural mutations. To illustrate, if the SSDL mutagen is applied to the code fragment in Figure 7, it will generate the four mutations displayed in Figure 8.

```
while (X != Y)
    if (X < Y)
        Y = Y - X;
    else
        X = X - Y;
```

Figure 7. Original while statement.

(i)	(ii)	(iii)	(iv)

`;`	`while (X != Y)`
	`    ;`
		`while (X != Y)`
		`    if (X < Y)`
		`        ;`
		`    else`
		`        X = X - Y;`
			`while (X != Y)`
			`    if (X < Y)`
			`        Y = Y - X;`
			`    else`
			`        ;`

Figure 8. The four SSDL induced mutations.

In this instance the entire WHILE statement must be metamutated in a way that embodies the four alternatives of Figure 8. It should be noted that such a metamutation is *always* possible, although the result may be ungainly, by enclosing all the alternatives in a case construct that selects which alternative to execute. This observation, in fact, is what gives us confidence that for any program and set of mutagens a corresponding metamutant can be generated. A concern, of course, is that the metamutant does not "bloat" to an extent as to make its compilation and execution prohibitively expensive. However the examples of expression metamutation previously given serve to show that is is possible to compactly represent many mutants without necessarily resorting to bulky code duplication within case alternatives. Addressing the specific situation above, each statement component is a change point; consequently enclosing each statement component in an IF statement that conditionally executes that component based on the value of the mutant descriptor provides a satisfactory way of implementing the metamutation.

Twinning

Executing any metamutated form of a statement will always be more expensive than executing the original statement given the *instantiation overhead*, that is, the cost of determining at run-time at each change point which alternative is to be used. *Twinning* is our strategy to improve metamutant execution performance wherein each statement actually appears twice, albeit in two forms: a *slow* fully metamutated form and a *fast* minimally modified form. The fully metamutated form of the statement is able to mimic the functionality of all the mutated alternatives of the statement but incurs the full cost of instantiation overhead. The other form of the statement incorporates only changes that allow the operands of the statement to reference the same address space as used by the meta-operand(s) and runs with virtually no extra overhead. These twin statements are collected into the THEN and ELSE clauses of an IF statement, respectively.

For example, applying the twinning strategy to the example `SUMSQRT` line 11 assignment statement

```
DELTA = GUESS - SQRT;
```

produces

```
if (8==Mutant_.StmtID)
    BO_(REF_(L_,8,2),OI_(BO_(OI_(REF_(L_,7,6),5),OI_(REF_(L_,6,8),7),15,4),3),37,1);
else
    L_->DELTA = L_->GUESS - L_->SQRT;
```

Recall from Chapter I that each mutant contains only one mutated statement: the remaining statements are identical to the original program. Hence when the metamutant runs representing the functionality of a mutant, the *slow twin* version of a statement is executed only if that statement contains the mutation. Otherwise the *fast twin* version of the statement is run. The performance benefits of this strategy are examined in Chapter IV.

Execution Accounting

An issue not directly related to metamutation but important in metamutant design is *execution accounting*. When running a mutant there is a chance that the mutation it contains will cause the mutant to enter an infinite loop. Since there is no general procedure for ascertaining if this has occurred, the usual way this is addressed in mutation analysis is to determine how much work the original program required on a test case and establish a limit that is some multiple of that work load [34]. Conventionally, this limit is ten times the original work load. When this limit is exceeded, the mutant is said to have *timed-out*. A time-out kills the mutant.

In conventional interpretive mutation analysis systems, the workload is measured as the number of statements interpreted. When running a mutant, a time-out occurs when the interpreter discovers that the number of statements interpreted exceeds the time-out limit.

The execution of a metamutant acting as a mutant must similarly be circumscribed. It is tempting to have the driver of the metamutant simply measure the amount of elapsed CPU time as a work load measure. The time the (metamutant acting as the) original program required on a test case would be recorded and the product of that value and some standard multiple would be used as a time-out limit. Before a mutant is executed, a system timer would be set to cause an interrupt at the time-out limit. If the metamutant had not finished execution when the timer interrupt was received, a time-out would occur. Unfortunately the disparity in the execution speeds of a fast twin statement and a slow twin statement makes such a scheme unusable. When a metamutant executes acting like the original program, it runs only fast twin statements since no mutated statements need to be executed. When a metamutant executes like a mutant, the percent of time spent running slow twin statements can vary widely depending on whether or not the mutated statement is in a frequently executed loop. Thus any simple timer-based scheme might result in spurious time-outs of slowly executing mutants where the mutated statement was within a deeply nested looping construct. Less serious, but a blow to performance, would be the delayed time-outs of mutants where the mutated statement was executed perhaps only once and the rest of the time the program was executing only fast twin statements.

To implement a usable time-out mechanism, we use a statement counting scheme similar in spirit to that used in interpretive systems. The major difference is that the metamutant itself must do the execution accounting in addition to its other duties of modeling mutants. Note that the metamutant must be designed to initially tally and record statement counts when running as the original program yet also be able to count statements executed and check for time-outs when running as a mutant. We do this by attaching two counters to each statement. One counter is a subscripted statement counter variable used to tally the number of times the individual statement was executed. The other counter is a "headway" counter that records the total number of statement executions; it is this counter that is checked for overflow (indicating a time-out) prior to executing any GOTO or upon entry to any loop body. For an illustration of this execution accounting mechanism, check the metamutant listed in Appendix A.

Design Specifics for C Mutagens

Any detailed discussion of metamutant design must necessarily consider a specific set of mutagens and the language to which they apply. The set we describe takes as its starting point the mutagens defined by Agrawal et al. for the C language in their report *"Design of Mutant Operators for the C Programming Language"* [51]; for sake of reference, this set of 82 mutagens[7] will be identified as G_1 and the reference document called the G_1 *report*. Several minor revisions and corrections are made to G_1 resulting in the 74 mutagen set we identify as G_2.

Not included in the mutagen counts above are the *category* mutagens. In both G_1 and G_2, most of the basic mutagens are aggregated into syntax-directed categories and subcategories. This leads to additional composite category mutagens. Thus, for example, the `Oneg` mutagen encompasses the basic mutagens `OLNG`, `OBNG`, and `OCNG`; applying the `Oneg` mutagen produces the same results as applying the `OLNG`, `OBNG`, and `OCNG` mutagens in concert. The mutagen naming conventions allow us to clearly distinguish between basic and composite mutagens: basic mutagen codes contain four upper-case letters whereas category mutagens start with an upper-case letter and end in three lower-case letters.

In standard mutation analysis, only a single syntactic change is made to a program to produce a mutant. Mutants that exhibit this property are called *first-order* mutants. Only first-order mutants populate the program neighborhoods in standard mutation analysis. Although the application of a mutagen to a program may result in more than one mutant being generated, proper mutagens induce only a single change per mutant. Moreover, each change must be syntactically legal. Declarations, the address operator (`&`), format strings in input/output functions, function prototypes, function identifiers, and C preprocessor directives are not mutated. Although the arguments to a function call may be mutated, the type and number of arguments to the call cannot be changed.

---

[7] Actually only 80 mutagens are described in the report: however the omission of the *constant for scalar* operand replacement mutagens (`Vcsr`) was clearly an inadvertent mistake. Section 12.3, labeled *constant for scalar*, really describes *scalar for constant* replacements. Adding to the confusion is the fact that the report incorrectly states that 77 mutagens are defined.

We broadly classify mutagens as (1) operator, (2) operand, or (3) structural mutagens. Each class has different metamutant design characteristics.

Operator Mutagens

Operator mutagens are further classified as either binary or unary.[8] The *binary operator replacement* mutagens model the incorrect choice of a C binary operator within an expression. Table I lists the binary operators of C.

The `Obor` (née `Obom`)[9] category mutagen represents 40 basic mutagens. Using the operator classification from Table I, `Obor` is subdivided into mutagens that belong to two subcategories: *comparable operator replacement* (`Ocor`) and *incomparable operator replacement* (`Oior`). Within these subcategories, mutagens affect either the *non-assignment* or the *assignment* operators in C. Each basic mutagen within the `Obor` category systematically replaces a C operator in its domain by operators in its range. Tables II and III give the *domain* and *range* for all 40 basic mutagens, along with an example mutation.

Table I. Classification of binary operators in C.

Type	Category	Operators	Domain Code
Non-assignment	Arithmetic	`+ - * / %`	`A`
	Bitwise	`| & ^`	`B`
	Logical	`|| &&`	`L`
	Shift	`<< >>`	`S`
	Relational	`== != < <= > >=`	`R`
Assignment	Arithmetic	`+= -= *= /= %=`	`A`
	Bitwise	`|= &= ^=`	`B`
	Plain	`=`	`P`
	Shift	`<<= >>=`	`S`

[8] C's sole *ternary* operator, "`?:`", is not mutated since it would require two syntactic changes to create a syntactically legal mutation. The operands, however, are subject to mutation.

[9] Wherever a mutagen code has been changed from that given by Agrawal et al. [51], we use the word "née" followed by the old code. This uses the secondary meaning of the word "née" so that "`Obor` (née `Obom`)" should be read as "`Obor` (formerly called `Obom` by Agrawal et al.)". Several mutagen codes were revised to make them more mnemonic. Hence `Obor` for binary operator replacement.

Table II. `Obor` binary operator replacement mutagens—comparable.

<table>
<tr><td colspan="4">`Ocor` comparable operator replacement</td></tr>
<tr><td colspan="4">Non-assignment Type</td></tr>
<tr><td>Mutagen Code</td><td>Domain</td><td>Range</td><td>Example</td></tr>
<tr><td>OAAN</td><td>Arithmetic</td><td>Arithmetic</td><td>a+b → a-b</td></tr>
<tr><td>OBBN</td><td>Bitwise</td><td>Bitwise</td><td>a&b → a|b</td></tr>
<tr><td>OLLN</td><td>Logical</td><td>Logical</td><td>a&&b → a||b</td></tr>
<tr><td>ORRN</td><td>Relational</td><td>Relational</td><td>a<b → a>b</td></tr>
<tr><td>OSSN</td><td>Shift</td><td>Shift</td><td>a<<b → a>>b</td></tr>
<tr><td colspan="4">Assignment Type</td></tr>
<tr><td>Mutagen Code</td><td>Domain</td><td>Range</td><td>Example</td></tr>
<tr><td>OAAA</td><td>Arithmetic</td><td>Arithmetic</td><td>a+=b → a-=b</td></tr>
<tr><td>OBBA</td><td>Bitwise</td><td>Bitwise</td><td>a&=b → a|=b</td></tr>
<tr><td>OSSA</td><td>Shift</td><td>Shift</td><td>a<<=b → a>>=b</td></tr>
</table>

We have previously presented in this chapter our `MSG` approach to representing *all* the mutations induced by these `Obor` category mutagens in a metamutant whereby each binary operator in the original program is replaced by the `BO_` metaoperator. For example, replacing the two binary operators—the assignment operator, "`=`", and the minus sign, "`-`"—in the following statement

```
DELTA = GUESS - SQRT;
```

produces

```
BO_(REF_(L_,8,2),BO_(REF_(L_,7,6),REF_(L_,6,8),15,4),37,1);
```

Because of dynamic typing, the operands have also been replaced in this example by `REF_` metaprocedures; this operand replacement will be considered in detail in discussing the operand mutagens below.

The *unary operator replacement* mutagens model errors in the use of unary operators and conditions within an expression. The `Ouor` (née `Ouom`) category mutagen represents the five basic mutagens listed in Table IV. The table also shows a further subdivision into two subcategories: the `Oidr` (née `Oidm`) *increment/decrement* mutagens and the `Oneg` *unary negation* mutagens.

Table III. `Obor` binary operator replacement mutagens—incomparable.

`Oior` incomparable operator replacement						
Non-assignment Type						
Mutagen Code	Domain	:	Range	Example		
`OABN`	Arithmetic	:	Bitwise	`a+b`	→	`a&b`
`OALN`	Arithmetic	:	Logical	`a+b`	→	`a&&b`
`OARN`	Arithmetic	:	Relational	`a+b`	→	`a<b`
`OASN`	Arithmetic	:	Shift	`a+b`	→	`a<<b`
`OBAN`	Bitwise	:	Arithmetic	`a&b`	→	`a+b`
`OBLN`	Bitwise	:	Logical	`a&b`	→	`a&&b`
`OBRN`	Bitwise	:	Relational	`a&b`	→	`a<b`
`OBSN`	Bitwise	:	Shift	`a&b`	→	`a<<b`
`OLAN`	Logical	:	Arithmetic	`a&&b`	→	`a+b`
`OLBN`	Logical	:	Bitwise	`a&&b`	→	`a&b`
`OLRN`	Logical	:	Relational	`a&&b`	→	`a<b`
`OLSN`	Logical	:	Shift	`a&&b`	→	`a<<b`
`ORAN`	Relational	:	Arithmetic	`a<b`	→	`a+b`
`ORBN`	Relational	:	Bitwise	`a<b`	→	`a&b`
`ORLN`	Relational	:	Logical	`a<b`	→	`a&&b`
`ORSN`	Relational	:	Shift	`a<b`	→	`a<<b`
`OSAN`	Shift	:	Arithmetic	`a<<b`	→	`a+b`
`OSBN`	Shift	:	Bitwise	`a<<b`	→	`a&b`
`OSLN`	Shift	:	Logical	`a<<b`	→	`a&&b`
`OSRN`	Shift	:	Relational	`a<<b`	→	`a<b`
Assignment Type						
Mutagen Code	Domain	:	Range	Example		
`OABA`	Arithmetic	:	Bitwise	`a+=b`	→	`a&=b`
`OAEA`	Arithmetic	:	Plain	`a+=b`	→	`a=b`
`OASA`	Arithmetic	:	Shift	`a+=b`	→	`a<<=b`
`OBAA`	Bitwise	:	Arithmetic	`a&=b`	→	`a+b`
`OBEA`	Bitwise	:	Plain	`a&=b`	→	`a=b`
`OBSA`	Bitwise	:	Shift	`a&=b`	→	`a<<=b`
`OEAA`	Plain	:	Arithmetic	`a=b`	→	`a+b`
`OEBA`	Plain	:	Bitwise	`a=b`	→	`a&=b`
`OESA`	Plain	:	Shift	`a=b`	→	`a<<=b`
`OSAA`	Shift	:	Arithmetic	`a<<=b`	→	`a+b`
`OSBA`	Shift	:	Bitwise	`a<<=b`	→	`a&=b`
`OSEA`	Shift	:	Plain	`a<<=b`	→	`a=b`

Table IV. Ouor unary operator replacement mutagens.

Oidr increment/decrement replacement		
Mutagen Code	Description	Example
OPPR	"plus-plus" Replacement (née OPPO)	++a → --a (*or* a++)
OMMR	"minus-minus" Replacement (née OMMO)	--a → ++a (*or* a--)
Oneg unary negation mutation		
Mutagen Code	Description	Example
OLNG	Logical Negation	a&&b → !a&&b
OBNG	Bitwise Negation	a&b → ~a&b
OCNG	Logical Context Negation	if(*expr*) → if(!*expr*)

The unary operators in C include the postfix and prefix increment operators (++), the postfix and prefix decrement operators (--), the unary minus (-), the unary plus (+), the bitwise complement (~), the logical complement (!) and the sizeof operator. Under our approach, to support the mutations induced by the Ouor category mutagens all unary operators in C are replaced by the UO_ metaprocedure. For example,

```
++NUM;
```

produces

```
UO_(REF_(L_,5,10),25,9);
```

The UO_ metaprocedure is implemented as a function with a prototype of

```
tREF_ UO_(tREF_ Expression, int DefaultOp, int ChangePoint);
```

At run-time, the actual unary operation performed is determined by the value of the mutant descriptor being processed.

Operand Mutagens

Although the operand mutagens in Table V are further classified by Agrawal et al. as either *variable* or *constant*, for our purposes it is more useful to classify them as either inducing operand replacements or inducing operator insertions.

Table V. Operand mutagens.

Variable mutations			
Mutagen			Description
Category	Sub-category	Code	
Vsrr	Scalar Reference Replacement		
	Vssr	Scalar for Scalar Replacement	
		VGSR	Vsrr using Globals
		VLSR	Vsrr using Locals
	Vcsr	Constant for Scalar Replacement	
		VGCR	Vcsr using Globals
		VLCR	Vcsr using Locals
Varr	Array Reference Replacement		
		VGAR	Varr using Globals
		VLAR	Varr using Locals
Vtrr	Structure Reference Replacement		
		VGTR	Vtrr using Globals
		VLTR	Vtrr using Locals
Vprr	Pointer Reference Replacement		
		VGPR	Vprr using Globals
		VLPR	Vprr using Locals
VSCR		VSCR	Structure Component Replacement
Vdom	Domain Mutations		
		VDTR	Domain Trap
		VTWD	Domain Twiddle

Constant mutations			
Mutagen			Description
Category	Sub-category	Code	
Ccrr	Constant Reference Replacement		
	Cscr	Scalar for Constant Replacement	
		CGSR	Cscr using Globals
		CLSR	Cscr using Locals
	Cccr	Constant for Constant Replacement	
		CGCR	Cccr using Globals
		CLCR	Cccr using Locals

Operand Replacement. The *operand replacement* mutagens model errors in the use of scalar references within a program. *Scalar references* are expressions that refer to a scalar value and can be simple constants, variables, or operational expressions that evaluate to an address. For example, the scalar references in the `FOO` function listed in Figure 9 are: `QTY`, `ANSWER`, `INDEX`, `AR`, `5`, `0`, `1`, `AR[INDEX++]`, `*ANSWER`, and `AR[INDEX-1]`.

```
void FOO(int QTY, int *ANSWER)
{
    int INDEX;
    float AR[5];

    INDEX = 0;
    while (INDEX < QTY)
        AR[INDEX++] = *ANSWER;
    AR[INDEX-1] = 0;
}
```

Figure 9. `FOO` function.

Our `MSG` approach to representing *all* the operand replacement mutations in a metamutant involves replacing each operand by a metaoperand function, called `REF_`, that is able to evaluate to any of the program's references. This metaoperand returns a value of type "generic data object", or `tREF_`, as described previously when dynamic typing was introduced.

In languages that support nested procedures, the most natural implementation of the metaoperand function would be as a procedure local to the metamutant procedure. Consequently, the metaoperand function would have full access to the variables in the metamutant's scope.[10] However C does not allow nested procedures and another mechanism must be used.

The approach we use is to create an addressable block of memory that contains all elements of the metamutant's local referencing environment and can be made accessible to

[10] In an early handcrafted proof-of-concept metamutant written in Modula-2, this is indeed how the metaoperator was implemented.

the REF_ metaoperand. To create this block, we move all declarations *inside* the metamutant to a structure type declaration that is *outside* the metamutant and is available to (by being placed physically before) both the REF_ and metamutant procedures. This type declaration, which we call LOCAL_, is then used within the metamutant to declare the local referencing environment. All references to items inside the block, which we call ENV_, are made via a pointer, called L_, to the block. Passing this pointer L_ as a parameter to the REF_ metaoperand function gives it full access to the variables in the metamutant's scope. This pointer L_ is also used inside fast twin statements when referencing variables: in this way the slow twin and fast twin statements can access and manipulate the same referencing environment.

To illustrate this local referencing environment mechanism, consider again the FOO function of Figure 9. The corresponding LOCAL_ type declaration, containing the declarations removed from inside of FOO, is shown in Figure 10. Notice that the formal arguments of FOO are included as part of the local referencing environment and are thus part of the LOCAL_ type declaration.

```
typedef
    struct
    {
        int QTY;
        int (*ANSWER);
        int INDEX;
        float AR[5];
    }
LOCAL_;
```

Figure 10. FOO's LOCAL_ type declaration.

Inside the metamutant of FOO, the ENV_ block is declared to be of type LOCAL_. Before use, the ENV_ block is cleared (i.e., set to zeroes), the formal parameters to FOO are copied into the block, and the block pointer, L_, is set. These actions are illustrated in Figure 11 which lists the beginnings of the FOO metamutant procedure. (Also illustrated in Figure 11 is the use on line 18 of the pointer L_ in a fast twin statement.)

```
void FOO( int QTY_PARM_, int (*ANSWER_PARM_) )
{    LOCAL_  ENV_;
     LOCAL_  *L_ = &ENV_;

     /* "Zero" the environment */
     (void) memset(&ENV_, 0, sizeof ENV_);

     /* Formal parameters => local equivalents */
     L_->QTY = QTY_PARM_;
     L_->ANSWER = ANSWER_PARM_;

     /* Begin FOO executable code */
     if (6!=Mutant_.ChangePoint)
     {   /*BEGIN 1*/
         if (1==Mutant_.StmtID)
             BO_(REF_(L_,5,10),REF_(L_,8,11),37,9);
         else
             L_->INDEX = 0;
     }   /*END 1*/
     . . .
```

Figure 11. The beginning of the `FOO` metamutant.

The referencing of global variables by `REF_` is straightforward: since such variables are visible to the `REF_` function, they are addressed directly through their identifier names.

To handle references to constants, static variables initialized to the values of the constants are declared within the `REF_` function. These static variables are then addressed by whatever identifier name they were assigned. Although there is some bookkeeping involved in determining all the program constants[11] and creating corresponding properly typed static variables, this is also a fairly straightforward process. Figure 12 lists `FOO`'s `REF_` metaoperand. The three program constants, `5`, `0`, and `1`, are represented in that metaoperand by the variables `const7_`, `const8_`, and `const9_`.

When the `REF_` metaoperand is invoked, the specific reference returned is selected in a case construct based on the value of the current mutant descriptor. Referring to Figure 12, each "`case`" in the "`switch`" statement is a possible program reference.

[11]Besides scanning the program for constants actually appearing in the original program, it might be necessary to add additional constants. In the mutagen set G_1, there is defined a *required constant replacement* (`CRCR`) mutagen that replaces scalar references by the constants zero, one, and negative one. We omit this mutagen in G_2 and allow the mutation analysis system the freedom of optionally adding these constants to those already in the program; it is this, possibly augmented, set of constants that is used by the *constant for scalar replacement* (`Vcsr`) and *constant for constant replacement* (`Cccr`) mutagens in performing substitutions and consequently the same mutations are induced.

```
tREF_  REF_(LOCAL_ * L_, int Original, int ChangePoint)
{
    tREF_     ref;
    int       Reference;

    static tINT_  const7_ = 5;
    static tINT_  const8_ = 0;
    static tINT_  const9_ = 1;

    if (ChangePoint==Mutant_.ChangePoint)
        Reference = Mutant_.Variation;
    else
        Reference = Original;

#define SETREF(ID,TYPE,INDR)  ref.addr = (tPTR_) &ID; \
  ref.type = TYPE;  ref.indr = INDR;
#define SETARR(ID,TYPE,INDR)  ref.addr = (tPTR_) &Result_[ChangePoint]; \
  Result_[ChangePoint].PTR_ = (tPTR_) &ID; \
  ref.type = TYPE;  ref.indr = INDR;

    switch (Reference)
    {
    case   3:  SETREF( L_->QTY,                 INT_,   0 );             break;
    case   4:  SETREF( L_->ANSWER,              INT_,   1 );             break;
    case   5:  SETREF( L_->INDEX,               INT_,   0 );             break;
    case   6:  SETARR( L_->AR,                  FLT_,   1 );             break;
    case   7:  SETREF( const7_,                 INT_,   0 );             break;
    case   8:  SETREF( const8_,                 INT_,   0 );             break;
    case   9:  SETREF( const9_,                 INT_,   0 );             break;
    case  10:  SETREF( L_->AR[(L_->INDEX++)],   FLT_,   0 );             break;
    case  11:  SETREF( *L_->ANSWER,             INT_,   0 );             break;
    case  12:  SETREF( L_->AR[(L_->INDEX - 1)], FLT_,   0 );             break;
    case  13:  ref = BO_(REF_(L_,6,26),UO_(REF_(L_,5,28),23,27),19,25); break;
    case  14:  ref = UO_(REF_(L_,4,31),28,30);                          break;
    case  15:  ref = BO_(REF_(L_,6,39),BO_(REF_(L_,5,41),
                         REF_(L_,9,42),15,40),19,38);                   break;
    default:   ERROR_("Illegal Reference Variant");
               STOP_();                                                  break;
    }

#undef SETREF

    return ref;
}
```

Figure 12. FOO's REF_ metaoperand.

The most difficult issue in the design of the REF_ metaoperand is the handling of operational expressions that represent scalar references. Such operational expressions are a composite of references and expressions that may themselves be subject to replacement or mutation. For example, the composite reference AR[INDEX++] will have the subscript expression mutate to AR[QTY++], AR[INDEX-1], AR[INDEX--], and AR[INDEX] to list a few. Each of these mutations must be represented by a *single* use of the REF_ metaoperand. We achieve this multiplicity of representations by letting the REF_ function call itself recursively and by fully metamutating each component of a reference. Thus, in FOO's REF_ metaoperand, this case

```
case 13:  ref = BO_(REF_(L_,6,26),UO_(REF_(L_,5,28),23,27),19,25); break;
```

represents all the mutations of the composite reference AR[INDEX++]. Since it would be computationally expensive to use this fully metamutated form of the reference each time, our design uses a twinning scheme within the metaoperand to reduce evaluation overhead. Thus the following case, which is the fast twin equivalent of "case 13" above, is ordinarily used

```
case 10:  SETREF( L_->AR[(L_->INDEX++)],   FLT_,   0 );            break;
```

and the slower form used only when a mutation affecting that reference is present.

Operator insertion. The two *operator insertion* mutagens are VDTR and VTWD. The Odom category mutagen encompasses them both. Although grouped by Agrawal el al. under the operand mutagens, these two mutagens affect entire expressions and perhaps belong in a grouping entirely of their own.

The *domain trap* (VDTR) mutagen provides a form of domain coverage, where the domain is subdivided into three subdomains: negative values, zero, and positive values. Each scalar expression E is mutated to TON_(E), TOZ_(E), and TOP_(E), where the semantics of TON_, TOZ_, and TOP_ are given in Table VI.

Table VI. Functions inserted by VDTR mutagen.

Function	Description	
TON_	"Trap On Negative"	Mutant killed if argument is negative, else return argument value.
TOZ_	"Trap On Zero"	Mutant killed if argument is zero, else return argument value.
TOP_	"Trap On Positive"	Mutant killed if argument is positive, else return argument value.

The *twiddle* (VTWD) mutagen models errors where the desired value of an expression is off by some small amount. Twiddle induced mutations are useful for checking boundary conditions. Each scalar expression E is mutated to SUCC_(E) and PRED_(E), where SUCC_ returns the immediate successor to the argument's value and PRED_ returns the immediate predecessor of the argument's value. In other words, SUCC_ returns the argument value minus ϵ and PRED_ returns the argument value plus ϵ, where ϵ is one for integer arguments and some fraction of the absolute value of the argument for floating point arguments.

Both of these two mutagens cause the *Operator-Insertion* abstract entity to be inserted at implicit change points just before all the expressions in a program. (An individual scalar reference is considered an expression.) This abstract entity is syntactically realized using the OI_ operator insertion metaprocedure. If no mutation is present, the OI_ function simply passes its argument value through; otherwise the OI_ metaprocedure takes on the semantics of one of the functions TON_, TOZ_, TOP_, SUCC_, or PRED_.

Casting. Peripherally related to the operand mutagens, but not dealing with any mutation, are the dual issues of *entering* and *departing* the dynamic type system. Extending the original meaning of the word, we speak of *"casting"* a value into or out of the "generic" data type tREF_ . This casting is necessary wherever the code being metamutated may invoke functions not subject to mutation, such as library functions, or where the expression in a RETURN statement must evaluate to a specific type.

The *Cast-To-Generic* abstract entity is inserted at (implicit) change points before function expressions whose values must be entered into the dynamic type system. These abstract entities are then substituted by one of thirteen different `ref`*XXX*`_` function calls that differ only in the type of argument they accept. For example, the statement

```
sqrt(x) + 100.0;
```

would be metamutated to

```
BO_(refDBL_(sqrt(valDBL_(REF_(L_,9,15))),11),REF_(L_,10,17),14,10);
```

where the `refDBL_` function casts the return value of `sqrt` into a form that can be used an operand to the `BO_` metaoperator.

The *Cast-To-Type* abstract entity is inserted at (implicit) change points before expressions whose values are constrained to be a specific data type. These abstract entities are then substituted by one of thirteen different `val`*XXX*`_` function calls that return one of the thirteen C data types. (Composite types are returned as pointers.) For example, the `pow` library routine has the function prototype

```
double pow(double x, int y);
```

and takes two arguments, a double precision floating point number x and an integer power y, and returns the double precision floating point value x^y. An invocation of the `pow` function would require the first operand to evaluate to a value of type `double` and the second operand to evaluate to a value of type `int`. The metamutated statement

```
pow(valDBL_(REF_(L_,8,14)),valINT_(REF_(L_,9,17)));
```

will properly cast the operands, where `valDBL_` returns a value of type `double` and `valINT_` returns a value of type `int`.

Structural Mutagens

The *structural* mutagens are listed in Table VII. These mutagens mutate entire statements and may induce changes to the very structure of the program. Using the same order as Table VII, each mutagen will be briefly described and then our `MSG` approach to representing the mutations induced by the mutagen will be presented.

Table VII. Structural mutagens.

All statements	
STRP	Trap on Statement Execution
SSDL	Statement Deletion
SMVB	Move Brace Up or Down
Jump statements	
SGLR	"goto" Label Replacement
SCRB	"continue" Replacement by "break"
SBRC	"break" Replacement by "continue"
Iterative statements	
SWRD	"while" Replacement by "do-while"
SDRW	"do-while" Replacement by "while"
SMTT	Multiple Trip Trap
SMTC	Multiple Trip Continue
Selection statements	
STRI	Trap on "if" Condition
SSWM	Switch Statement Mutation

STRP. The *trap on statement execution* (STRP) mutagen systematically replaces each statement by a "TRAP_" statement that, if executed, kills the mutant. This mutagen is intended to assure program statement coverage.

Rather than individually executing such mutants, it is more efficient to use the metamutant execution accounting (described previously). Using the statement counters within the metamutant, it is possible to determine whether or not a statement has been reached and consequently whether the corresponding "TRAP_" statement has triggered a mutant kill.

SSDL. The *statement deletion* (SSDL) mutagen, which systematically deletes each statement of the program, was already discussed on page 21.

SMVB. The *move brace up or down* (SMVB) mutagen is initially described in the G_1 report as modeling errors in the placement of the terminating brace in a compound statement, hence its name. Later comments and examples in the G_1 report make it clear that this is a misnomer and that the mutagen is meant to model errors of commission and omission in loop bodies. Thus this mutagen generates two mutations per loop construct (no matter if the loop body is a compound statement with braces or a simple statement without braces): a statement immediately following a loop body is pushed inside the body (i.e., “move brace down”) and the last statement inside the loop body is pushed out of the body (i.e., “move brace up”).

To accommodate these mutations within a metamutant requires for each loop that the final statement of the loop body and the first statement following a loop be cloned and placed outside and inside the loop, respectively. These cloned statements are guarded by IF statements. Figure 13 illustrates this transformation for a WHILE loop. The predicates “MVB down” evaluate to TRUE if the statement immediately following a loop body is to be pushed inside the body and the predicates “MVB up” evaluate to TRUE if the last statement inside the loop body is to be pushed out of the body; otherwise these predicates normally evaluate to FALSE.

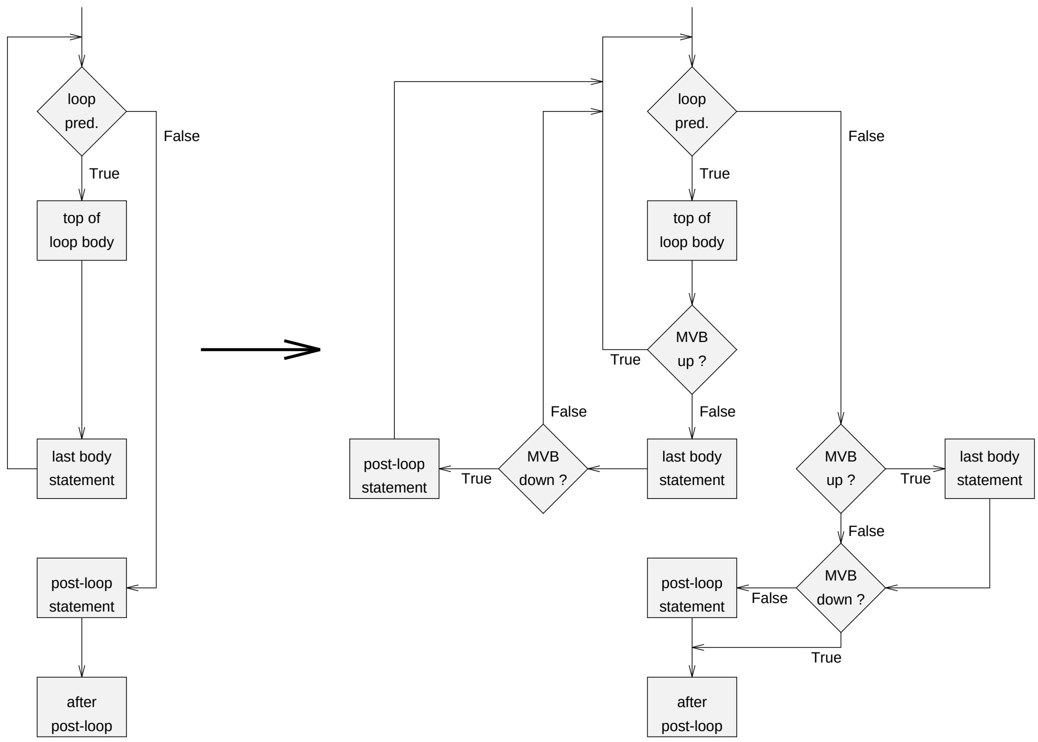

Figure 13. Transformation to support `SMVB` mutants.

<u>SGLR.</u> The "goto" *label replacement* (SGLR) mutagen models errors in specifying the destination label of a goto statement. Suppose a program has six labels, then the SGLR mutagen will induce five mutations of the statement "goto L101", where each mutated goto statement will branch to one of the other five labels.

In our approach, "goto" statements are replaced by the GOTO_ metaprocedure. The GOTO_ metaprocedure is a macro that is generated along with the metamutant and provides a selection of goto variants. In our example of a program with six labels, this GOTO_ metaprocedure would be used.

```
#define GOTO_(org,cp) switch \
  (cp==Mutant_.ChangePoint?Mutant_.Variation:org) { \
case  18:  goto L101; \
case  22:  goto L102; \
case  23:  goto L103; \
case  24:  goto L104; \
case  27:  goto L105; \
case  28:  goto L106; \
}
```

<u>SCRB.</u> The "continue" *replacement by* "break" (SCRB) mutagen models the erroneous substitution of a "break" for a "continue" and generates one mutation per "continue" statement.

Within the metamutant, we substitute a CONTINUE_ metaprocedure for each occurrence of a "continue" statement in order to emulate this mutation. This CONTINUE_ metaprocedure is implemented as a macro.

```
#define CONTINUE_(org,cp) {if (cp!=Mutant_.ChangePoint) \
    continue; \
else          \
    break;}
```

<u>SBRC.</u> The "break" *replacement by* "continue" (SBRC) mutagen models the erroneous substitution of a "continue" for a "break" and generates one mutation per "break" statement.

Within the metamutant, we substitute a BREAK_ metaprocedure for each occurrence of a "break" statement in order to emulate this mutation. This BREAK_ metaprocedure is implemented as a macro.

```
#define BREAK_(org,cp) {if (cp!=Mutant_.ChangePoint) \
    break;    \
else          \
    continue;}
```

<u>SWRD.</u> The "`while`" *replacement by* "`do-while`" (`SWRD` née `SWDD`) mutagen models the error of a WHILE statement being replaced by a DO-WHILE statement.

In our approach, WHILE statements of the form "`while(`*predicate*`)`" are metamutated to the form "`while(LOOP_(0,cp)||`*predicate*`)`". If the loop is to behave as a WHILE loop, the `LOOP_` procedure always returns FALSE. If the loop is to behave as a DO-WHILE loop, the `LOOP_` procedure returns TRUE the first time through and FALSE thereafter. The short-circuit evaluation semantics of the logical OR, "`||`", assure that the *predicate* is evaluated only when it should be.

<u>SDRW.</u> The "`do-while`" *replacement by* "`while`" (`SDRW` née `SDWD`) mutagen models the error of a DO-WHILE statement being replaced by a WHILE statement.

In our approach, DO-WHILE statements of the form "`do {...} while(`*predicate*`)`" are metamutated to the form "`while(LOOP_(1,cp)||`*predicate*`) {...}`". If the loop is to behave as a DO-WHILE loop, the `LOOP_` procedure returns TRUE the first time through and FALSE thereafter. The short-circuit evaluation semantics of the logical OR, "`||`", assure that the *predicate* is evaluated only when it should be. If the loop is to behave as a WHILE loop, the `LOOP_` procedure always returns FALSE.

<u>SMTT.</u> The *multiple trip trap* (`SMTT`) mutagen causes a mutation that provides a type of program instrumentation. The `SMTT` mutagen introduces a guard in front of loop bodies. This guard is a boolean function named `TrapAfterNthTrip`; when this function is evaluated the Nth time through the loop it kills the mutant. The value of N is decided by the tester.

The metamutation required to support this mutation is the same as that required for `SMTC` mutations and is described in the discussion of the `SMTC` mutagen that follows.

SMTC. The *multiple trip continue* (SMTC) mutagen introduces a guard in front of loop bodies. This guard is a boolean function named FalseAfterNthTrip. During the first N iterations of the loop, this function evaluates to TRUE, thus letting the loop body execute. After the first N iterations of the loop, this function evaluates to FALSE, thus causing the loop to iterate sans the body. The value of N is decided by the tester. This bizarre mutagen can be better understood using an example given in the G_1 report, where the following FOR statement

```
for (i=left+1; i<=right; i++)
{
    ...loop body...
}
```

is mutated to become

```
for (i=left+1; i<=right; i++)
    if (FalseAfterNthTrip())
    {
        ...loop body...
    }
```

The MSG-based design of a metamutation to incorporate the mutations induced by the SMTT and SMTC mutagens follows the mutation structure closely. The entire body of a loop is enclosed in an IF statement containing the BODYGUARD_ boolean function as its predicate. This boolean metaprocedure, which accepts a single change point argument, returns TRUE or behaves like either the TrapAfterNthTrip or the FalseAfterNthTrip guards depending on the value of the mutant descriptor.

STRI. The *trap on* "if" *condition* (STRI) mutagen is designed for providing IF statement branch analysis. Each IF statement of the form "if (*predicate*)" is mutated to "if (TOT_(*predicate*))" and "if (TOF_(*predicate*))", where the semantics of TOT_ and TOF_ are given in Table VIII.

In our approach, the predicate must be cast from a "generic" data type into a boolean (really an integer) value. The PRED_ metaprocedure does this casting. Additionally, PRED_ can behave as the TOT_ and TOF_ functions, as well as negating the predicate as required by the OCNG mutagen, depending on the value of the mutant descriptor.

Table VIII. Functions inserted by STRI mutagen.

Function	Description	
TOT_	"Trap On TRUE"	Mutant killed if argument is TRUE, else return FALSE.
TOF_	"Trap On FALSE"	Mutant killed if argument is FALSE, else return TRUE.

SSWM. The *switch statement mutation* (SSWM) mutagen creates mutants that are intended to provide SWITCH statement case coverage analysis. Conceptually, a SWITCH statement of the form "switch (*expr*)" with n different case labels is mutated to n different "switch (TOCi_(*expr*))" alternatives, where TOCi_ stands for "Trap on Case i ". An additional mutation of the form "switch (TOCD_(*expr*))" is also generated, where TOCD_ stands for "Trap on Case Default".

Although the description of the SSWM mutations suggests a change to the SWITCH statement header, in our approach it is the "case" labels that are modified in the metamutant. Each "case" and "default" label has appended to it a NULL statement. The metamutant execution accounting is able to determine which of these NULL statements were executed and thus is easily able to ascertain the degree of case coverage.

Invalid G_1 Mutagens

The G_1 reports describes several mutagens that, on closer examination, are not valid in the context of standard mutation analysis. Recall that a valid mutagen induces only a single syntactic change per mutant, that is, creates first-order mutants. Also, by definition, a mutant must be a syntactically correct program.

SBRN and SCRN. The SBRN mutagen replaces each "break" statement by a version that breaks out to the Nth enclosing level, where the value of N is decided by the tester. Similarly, the SCRN mutagen replaces each "continue" statement by a version that continues out to the Nth enclosing level. However no such multi-level BREAK or CONTINUE constructs exist in C. By adding a variety of labels to the program and cleverly using GOTO statements, such multi-level BREAKs and CONTINUEs could be simulated, however the resulting patchwork program would no longer be a first-order mutant. Since the `SBRN` and `SCRN` mutagens fail to produce a syntactically legal first-order mutant, they are invalid.

SRSR. The "`return`" *statement replacement* (`SRSR`) mutagen requires that each statement in the program be replaced by the various RETURN statements (and their associated return expressions) within the program. This rewriting of the program requires more than a single syntactic change and thus this mutagen is invalid.

SSOM. A sequence (or comma) expression consists of two or more subexpressions separated by a comma. The subexpressions are evaluated left-to-right; except for the rightmost subexpression, the values are discarded. Thus the statement "`r=(a,b,c,d);`" is semantically equivalent to "`a;b;d;r=d;`". For a sequence expression with N subexpressions, the *sequence operator mutation* (`SSOM`) mutagen produces $N-1$ mutations by rotating left the sequence one subexpression at a time. Shuffling subexpressions like this is akin to shuffling program statements—in either case the result is not a single syntactic change. Hence, this mutagen is invalid.

VASM. The *array reference subscript mutation* (`VASM`) mutagen causes the rotation of array subscripts within multidimensional array references, much like the `SSOM` mutagen rotated subexpressions. It is invalid for the same reason—it requires more than a single syntactic change to be made.

OIPM. Given the expression "`***x++`", the *indirection operator precedence mutation* (`OIPM`) operator would produce "`**(*x)++`", "`**++(*x)`", "`*(**x)++`", "`*++(**x)`", and "`++(***x)`" as mutations. Since these require more than a single syntactic change to achieve, the mutagen is invalid.

OCOR. The *cast operator replacement* (`OCOR`) mutagen is invalid in that it violates the principle that types are not to be mutated.

CRCR. The *required constant replacement* `CRCR` mutagen is not invalid as much as it is superfluous. As discussed on page 33, the operand mutagens will produce the mutations that the `CRCR` mutagen was designed to generate.

Automated Metamutant Generation

The metamutant concept would be of little use without an automated way to generate metamutants. The process we have developed 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 leaf node represents an operand and the non-leaf nodes represent either operators or structural information. A *decorated* AST has attributes, such as type information, attached to the nodes [53]. In the `MSG` method, we generate metamutants using operations centered on manipulating decorated abstract syntax trees.

An *attribute grammar* consists of a context-free grammar, a finite set of attributes, and a finite set of side-effect-free semantic rules. The `MSG` method uses an attribute grammar 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. (For the non-leaf nodes of the AST, type information is a *synthesized* attribute; that is, the data type of each node is determined from type information obtained from the children of the node. Type information flows "up" an AST.) Figure 14 shows a statement and its corresponding decorated AST.

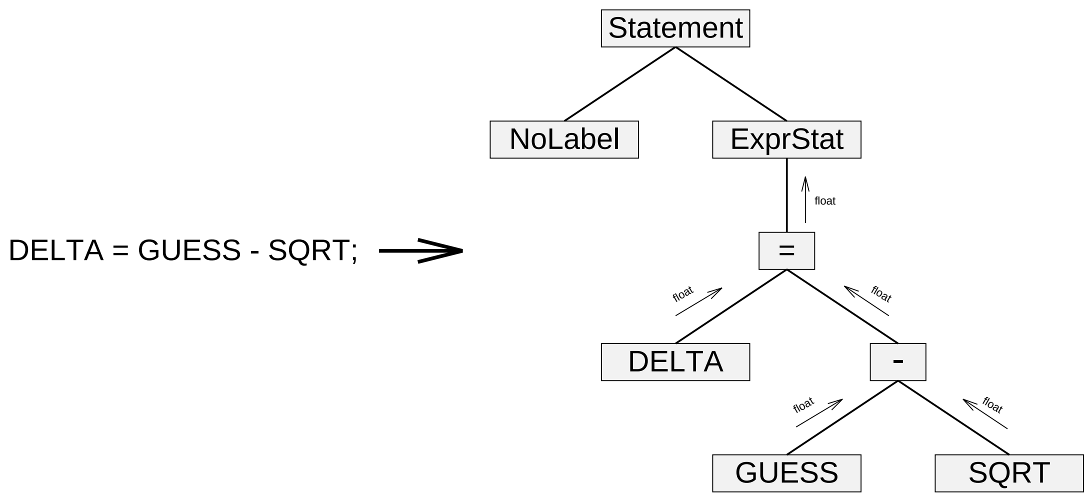

Figure 14. Statement and corresponding abstract syntax tree.

Mutagens are expressed as tree transformation and traversal 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 some node contents with metaprocedure references. Leaf nodes are replaced by metaoperands and interior nodes are replaced, where applicable, 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 along certain arcs. Additionally parts of the tree are duplicated and combined with IF statements to accommodate twinning. Figure 15 gives an example of such an AST transformation. By traversing the transformed AST, the information needed to generate a metamutant program is obtained. An AST traversal over the change points also provides the information needed to generate the list of mutant descriptors, D.

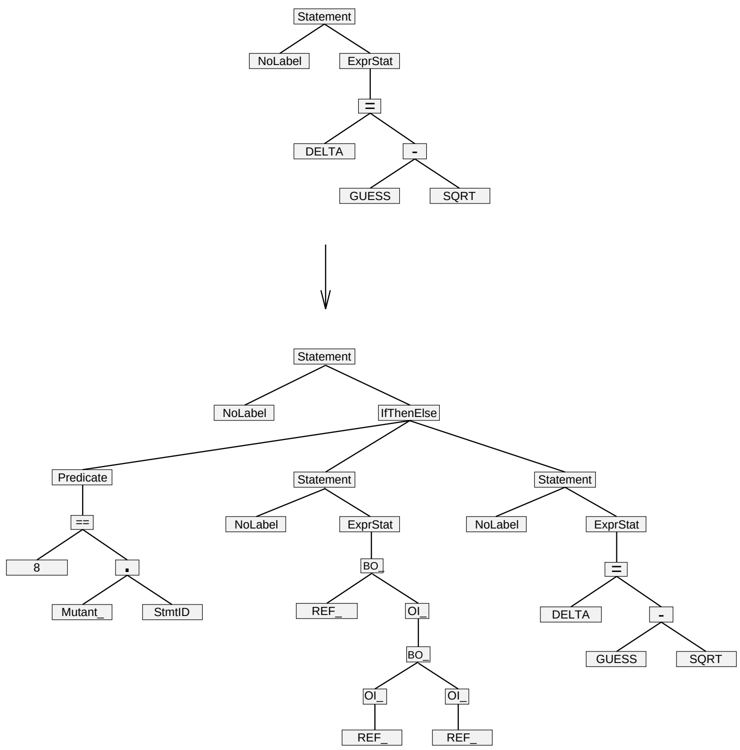

Figure 15. An example abstract syntax tree transformation.

Although conceptually the AST undergoes the transformation just described, for the sake of efficiency it is prudent to modify the tree as little as possible. Consequently in practice several of the transformations are implied and performed as needed by multiple AST traversals.

Chapter III contains further details on automated metamutant generation.

CHAPTER III

TUMS: A PROTOTYPE SYSTEM

This chapter provides an overview of our design and implementation of a prototype mutation analysis system based on the MSG method.

Goals of the System

The `TUMS` system, an acronym for Testing Using Mutant Schemata, was created to fulfill the following goals:

1. Demonstrate the automated generation of metamutants and mutant descriptors.
2. Gain experience in metamutant design and refine metamutant design heuristics.
3. Empirically study the performance of `MSG`-based mutation analysis systems.

The design of the `TUMS` system, being a research prototype expected to change and evolve, stressed simplicity and flexibility over efficiency. Although we were concerned that the metamutants generated by `TUMS` ran efficiently, that same degree of concern was not extended to the generation process. It was also not our intention to create a full-featured system that handled the full ANSI C language or every mutagen described in the previous chapter, but rather to create a representative `MSG`-based mutation analysis system with sufficient features for us to achieve the three goals listed above.

System Description

The `TUMS` system is designed around six central entities or objects. The set of tools that comprise the `TUMS` system exist to manipulate and interact with these objects. The objects are: P, the *program unit* being tested; T, the *test data* being analyzed; N, the *program neighborhood* of P; I, the *interface* between P and T; TS, the *test set*; and ART, the *analysis results table*.

The program unit or function, P, being tested is stored in a standard C source file. Via parameters, the user can specify which function within the source file is to be mutated.

The test data T is stored in a standard sequential text file. It is the mutation adequacy of this test data relative to the program unit P that the user wishes the mutation analysis system to determine and report as a mutation adequacy score. Unlike some other mutation analysis systems, such as `Mothra`, in `TUMS` the test data is *not* changed as a result of running a mutation analysis.

The program neighborhood N has two components: a metamutant M and a list of mutant descriptors D. The metamutant M is a syntactically valid C program with five sections as illustrated in Figure 16. The *prefix* section contains all the unmutated source material prior to the program unit P. Possibly empty, the prefix usually contains material that has been "`#include`d" via preprocessor statements. The `LOCAL_` and `REF_` sections were explained in the previous chapter. The *body* contains the metamutated statements of the program unit under test. The *suffix* section contains the unmutated source material, if any, that follows the program unit.

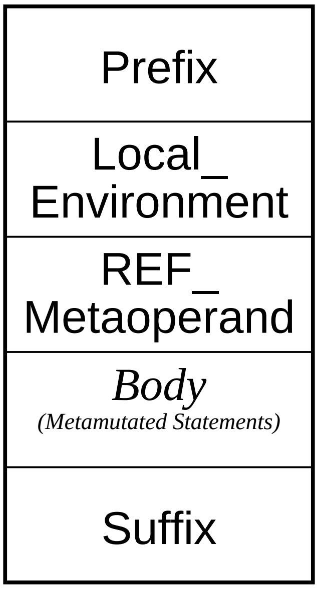

Figure 16. The five sections of a `TUMS` metamutant.

Each mutant descriptor contains the information necessary to instantiate the meta-mutant to function as a specific mutant, that is, it contains the parameter values that completely specify the desired mutant semantics. There are seven fields in the mutant descriptor: (1) a mutant id field, which is a serial number used to identify each mutant; (2) a mutagen name, which is the character string representation of the mutagen that induced this mutation (e.g., "`SSDL`"); (3) a numeric mutagen code; (4) the variation or alternative to the default action to be taken at the change point, expressed as a numeric code; (5) the change point number; (6) the number of the statement being mutated; and (7) a status code character, usually an "`L`" indicating the mutant descriptor should be processed but possibly an "`I`" indicating the mutant descriptor should be ignored.

Unlike some other systems that also have constructs called mutant descriptors, in particular `Mothra` [34] and `IMSCU` [52], mutant descriptors in `TUMS` are meant to record constant information and consequently *do not change as a result of mutant execution.* In particular, the status code is *not* used to record mutant mortality information (i.e., whether the mutant is dead). The status code field records properties of the mutant: "`E`" if the mutant is known to be equivalent, "`L`" if it is believed to be killable, and "`I`" if the experimenter wishes the mutant to be ignored.

The interface object I defines a relationship between a program unit P and some test data D. The interface contains information on: the number and type of parameters used to invoke the program unit P; whether these parameters send values to P, whether these parameters receive values from P, or both (that is, whether parameters are `IN`, `OUT`, or `INOUT`); how values received from P are to be compared to determine if a mutant has been killed (that is, specifying a mutant oracle); and which fields in the test data file correspond to the various parameters.

The interface object is a novel feature of `TUMS.` Other systems, such as `Mothra` and `IMSCU`, associate this information with the test data itself. In these other systems, the interface mapping must be entered anew each time a new test data file is prepared.

The test set object *TS* is comprised of *test case pairs* that contain both input and corresponding expected output. Although closely tied to the test data D, it is not the same as the test data. The test data contains only input and may contain fields not used in the testing of P; the test set contains only fields relevant to the testing of P as defined by some interface I and also contains the expected output that comes from running the original (unmutated) program plus a record of total statements executed in the original program. This expected output is compared, using the interface specifications in I, against the output that comes from running the mutant programs—if the outputs do not match properly, the mutant is killed.

The analysis results table object, *ART*, contains one row for each mutant and one column for each test case. The entries in the table indicate the status of the corresponding mutant when run against the corresponding test case. The bookkeeping resulting from mutation analysis is entered in this table. An entry of "`L`" indicates that the mutant survived this test case; an entry of "`D`" indicates that this test case caused the mutant to die (i.e., it differentiated the mutant from the original program); an entry of "`.`" indicates that no information about mutant mortality relative to the test case is available; and an entry of "`E`" or "`I`", probably transcribed from the mutant descriptor across the entire row, indicates that the mutant was not executed. An additional column is added to the table to record per mutant summary information. Thus if a mutant was killed by any test case, the summary column would show a "`D`". Usually once a "`D`" is entered in the summary column the mutant is not run against any further test cases (resulting in entries of "`.`" for these further test cases), however this default action can be overridden if it is desired to test the mutants with all the test cases. The *ART*, consequently, can record more than just the current mutant status that is usually the only item maintained by other mutation analysis systems.

Of the six data objects, only the analysis results table *ART* is updated while performing mutation analysis, that is, while executing mutants. This deliberate design decision not only speeds access to the other objects, since we do not have to write back to disk possible updates to these objects, but is meant to look forward to future distributed implementation of TUMS. The coarse granularity of mutation analysis, with each mutant execution being a candidate for a separate processor, makes a distributed implementation a natural next step. By isolating updates to this one data object, update problems are minimized and less information needs to be sent over the network.

Figure 17 graphically depicts as a data flow diagram the relationships between the central data objects and the TUMS tools. We next consider each of these tools in turn.

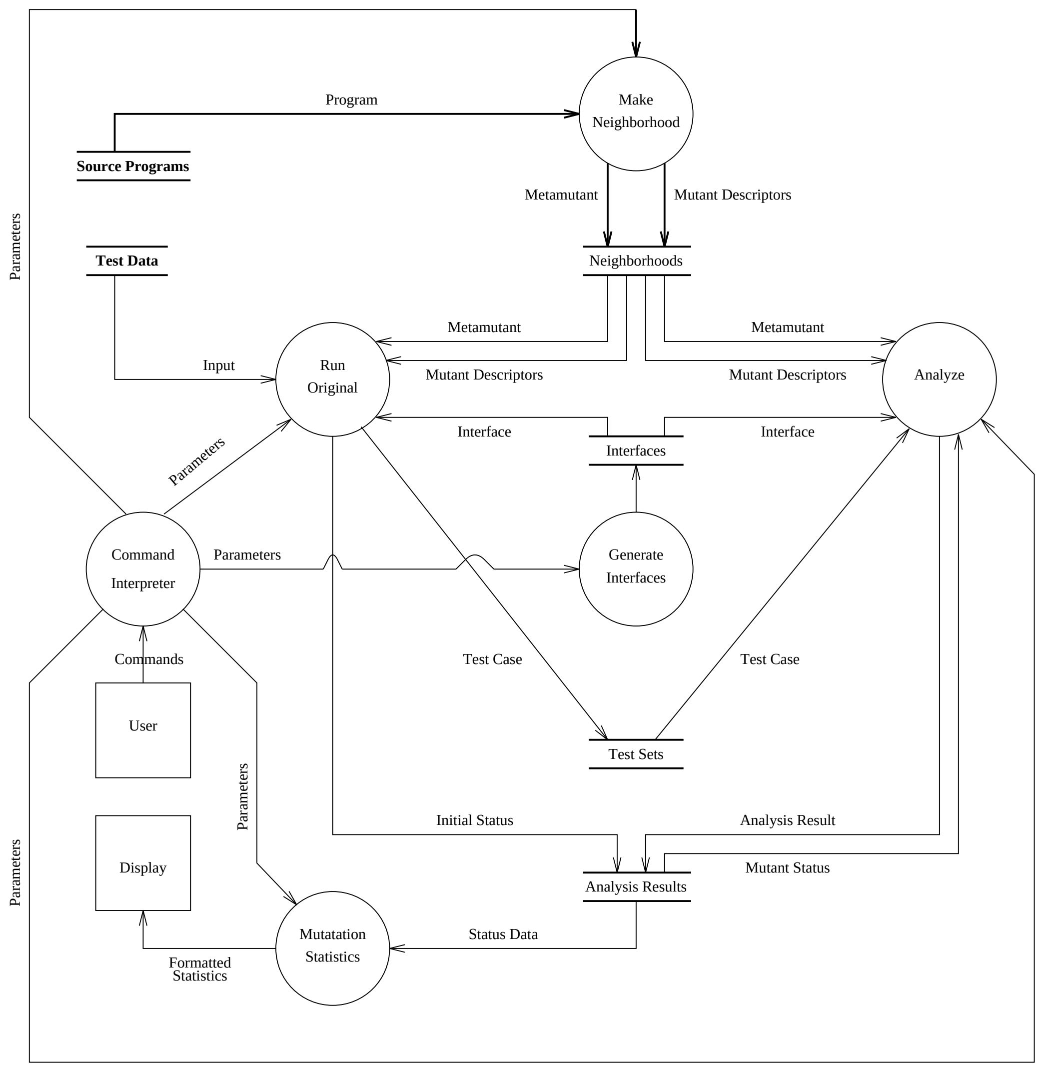

Figure 17. TUMS data flow diagram.

Make Neighborhood Tool

The `Make Neighborhood` tool takes a program unit P and generates the corresponding metamutant M and mutant descriptor list D. Parameters to the tool specify which mutagens to use, whether or not the `CRCR` required constants should be used in substitutions involving constants, and whether or not to generate fast twins. The internal data flow of the `Make Neighborhood` tool is shown in Figure 18.

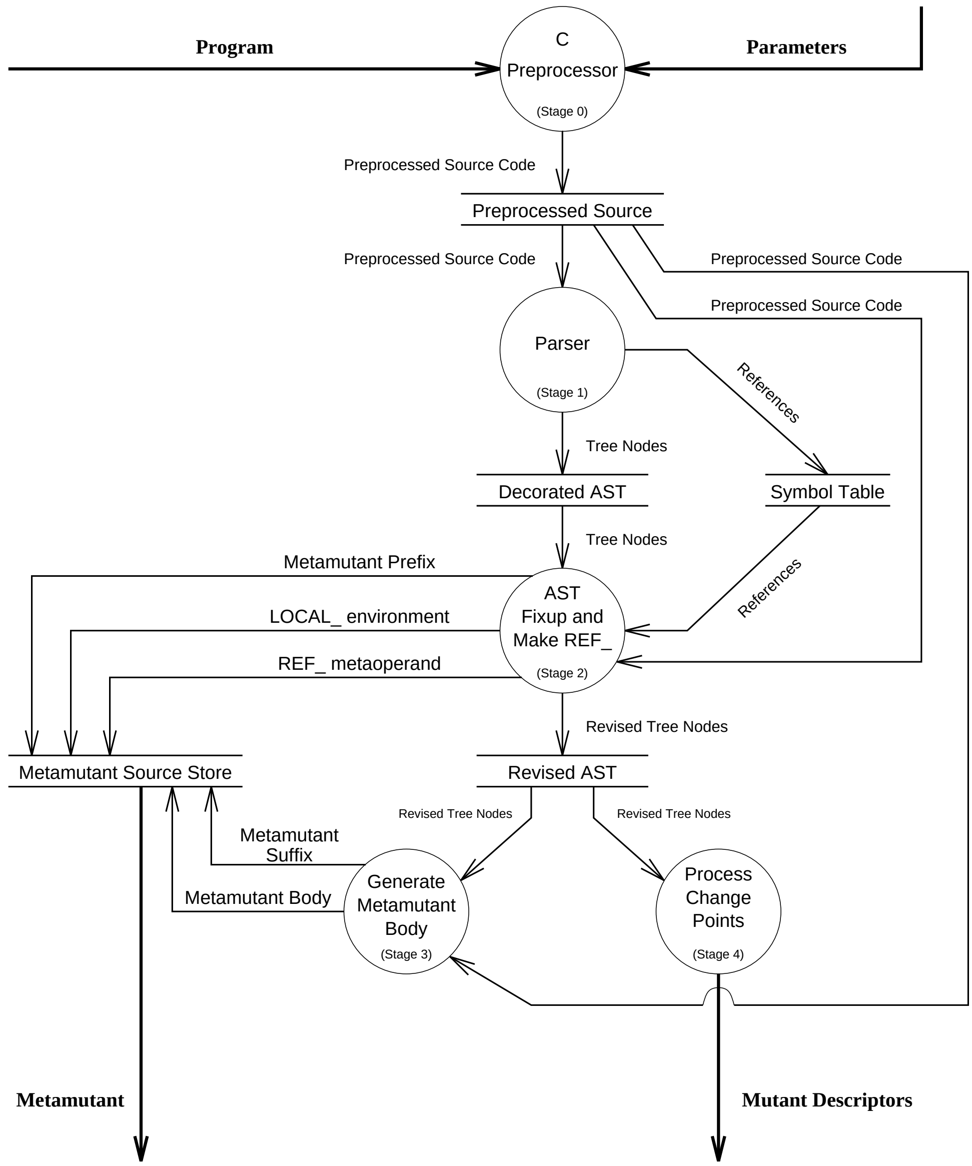

Figure 18. `Make Neighborhood` data flow diagram.

The Make Neighborhood tool processes its inputs in several stages. The Make Neighborhood tool begins, in stage 0, by invoking the GNU C preprocessor to handle "#includes" and macro definitions within the source program file. The resulting preprocessed source code is then sent to the parser.

In stage 1, the parser parses the preprocessed source code, gathers program references and creates a symbol table from these references, builds an abstract syntax tree (AST), and decorates the AST nodes with type information. It also marks the bounds within the preprocessed source code file of the function being mutated for use in the next stage.

The subset of ANSI C understood by the TUMS parser is rather large, corresponding roughly to the subset of C taught students in a "CS1" course. The chief omissions are typedefs and struct/unions; these elements would have greatly complicated the prototype's symbol table without contributing anything to the research goals of the system. Appendix I fully details the elements of ANSI C that are supported.

Stage 2 creates the metamutant prefix by copying to the metamutant file all the preprocessed source code that precedes the function being mutated. Using the symbol table information collected by the parser, this stage also emits the LOCAL_ type declaration. Based on the mutagens enabled, this stage transforms the decorated AST in the manner described in Chapter II. Leaf nodes are replaced by metaoperands and interior nodes are replaced, where applicable, by metaoperators. Some mutagens, such as the operator insertion mutagens, cause new nodes to be created corresponding to implicit change points. While this stage traverses and transforms the AST, it also collects the scalar program references. These collected scalar program references are used to generate the REF_ metaoperand described in Chapter II.

Once the transformed (or revised) AST is produced, stage 3 traverses the AST and produces the metamutant body. Each node encountered in the tree walk directs the generation of the code. It is at this stage that the abstract entities recorded in the transformed AST are turned into syntactically valid constructs. If twinning is enabled, sections of the

AST may be traversed twice: once to produce a fully metamutated version of a statement and again to reproduce a slightly modified version of the original. Stage 3 also copies all the preprocessed source code following the mutated function to the metamutant suffix section of the metamutant file.

Figure 18 shows that the transformed (or revised) AST is also used by stage 4 to produce the mutant descriptors. Each program change point, implicit or explicit, has a node in the transformed AST. The tree is again traversed. When a change point is reached, depending on the mutagens enabled, the alternatives at that point are determined and used to generate the corresponding mutant descriptors.

Recognizing that it was unnecessary to implement all of the G_2 mutagens in a prototype to demonstrate the viability of the `MSG` approach, a reduced, but representative, set G_3 was selected. The set G_3 is equal to the set G_2 less the mutagens `SMVB`, `SMTT`, `SMTC`, `SSWM`, `Vtrr`, and `VSCR`. (The last two mutagens are merely inapplicable since they deal with `structs`.) Every mutagen category is covered by one or more G_3 mutagens.

Most of the `Make Neighborhood` tool was specified using a combination of *attribute grammars* and *tree pattern matching rules*; these formalisms were taken from the `Cocktail` Compiler-Compiler Toolkit developed at the University of Karlsruhe in Germany [54]. The tools we used from the `Cocktail` toolkit include: `Ast`, `Rex`, `Lalr`, and `Puma`.

The `Ast` tool supports the definition and manipulation of attributed trees and graphs [55]. `Ast` accepts as input a description of a tree expressed in an extended BNF grammar notation and generates corresponding type structures and tree manipulating procedures. In the context of the `Make Neighborhood` tool, we use it to create an abstract syntax tree (AST) object type, complete with member functions or methods. These functions allow us to create attributed nodes of an AST, assemble them into a complete tree, and even check if they are structurally correct. We also use it to specify the symbol table as a graph object. `Ast` partially, but not totally, automates the low level implementation details involved in symbol table processing.

The Rex tool is used in constructing lexical scanners from specifications given as regular expressions [56]. Lalr is used to construct parsers from attribute grammars [57]. Recall that an attribute grammar consists of a context-free grammar, a finite set of attributes, and a finite set of side-effect-free semantic rules. The procedures created by the Ast tool can be used in expressing the semantic rules that are part of the attribute grammar. These three tools, Rex, Lalr, and Ast, used together generate the Make Neighborhood parser. For this reason, we stated in Chapter II that the MSG method uses an attribute grammar to direct both the parsing of the program and the AST construction. The hard part, of course, is writing a suitable attribute grammar.

Puma is a tool that supports the transformation and manipulation of attributed trees [58]. Puma requires a structural specification of the trees to be manipulated and a description of the transformations to be applied. The structure of the input trees is described by the same tree grammars used by the Ast tool. The desired tree transformations are specified using a set of tree pattern matching rules that are tied to semantic actions. In the context of the Make Neighborhood tool, we use Puma for three purposes: to generate the stage 2 component that "fixes up" the original AST, to generate the stage 3 tree walker that produces the metamutated source code, and to generate the stage 4 tree walker that produces the mutant descriptors. Mutagens in TUMS are thus expressed as Puma input specifications.

Expressing the "size" of the Make Neighborhood tool is difficult. The usual measures, number of statements and lines of code, are likely to be very misleading since much of that code was generated by Cocktail tools from the high level specifications that we wrote. Nonetheless, since no other obvious measures suggest themselves, we estimate that Make Neighborhood consists of approximately 12,400 C statements stored in files that contain approximately 35,000 lines. The C statements estimate comes from counting tokens that usually indicate the presence of a C statement: semicolons and the lexemes "if", "while", "for", and "switch". The lines estimate is a simple count of non-blank source lines.

The command "mn" invokes the Make Neighborhood tool.

Generate Interfaces Tool

The `Generate Interfaces` tool creates the five files that comprise the interface object I. These files are later used by the `Run Original` and `Analyze` tools in creating the "drivers" that invoke the metamutant. These files are

1. `VarDecl.h`, which contains the function prototype for P and declarations for the various arguments used to invoke P.
2. `Call.c`, which contains the C source statement text used to invoke P.
3. `Compare.c`, which contains the C source code that describes how to compare the before and after argument values to determine if the mutant should be killed.
4. `ReadTC.c`, which contains the C source code for reading the test case records from the test set file TS.
5. `WriteTC.c`, which contains the C source code for writing the test case records to the test case file TS.

Appendix B contains an example of these files for use with the `SUMSQRT` program.

In the `TUMS` prototype, the format of the test data file D and the test set file TS are the same and thus the `ReadTC.c` file can be used to read both files. If the file formats are different, then a sixth file containing the C source code for reading test data records from D would be required.

Since each interface represents the intersection of a program unit P with some test data D, the interface files are stored in a sub-directory whose name reflects this intersection.

Originally an editor-like program was created to generate these files. However it proved easier to create and modify the files directly with a standard text editor (e.g., `vi`) and the editor-like program was scrapped. Currently the `Generate Interfaces` tool is a 39 line C-shell script that merely creates the sub-directory in which the interface files are to be stored and copies some example files into the sub-directory to serve as editing templates.

The command "`gi`" invokes the `Generate Interfaces` tool.

Run Original Tool

The `Run Original` tool executes the original (unmutated) program against the test data file D and uses the results of that execution to create the test set file TS.

This tool also creates and initializes the analysis results table ART. Most entries in the ART will be set to "`.`", denoting no mutant mortality information is available. However using the statement execution tallies that were created as a result of running the original program, this tool can determine which *trap on statement execution* `STRP` mutants have been killed simply by checking which statements have been reached. (In future versions of `TUMS`, the status of the *switch statement mutation* `SSWM` mutants could be determined in the same way.) Additionally, for those mutants whose mutant descriptor status field contains "`I`" (ignore) or "`E`" (equivalent), this information is transcribed across all of that mutant's test case columns, effectively exempting that mutant from ever being instantiated and executed.

Each time the `Run Original` tool is used with a new program P or a new interface I, it creates a "driver" routine capable of invoking the metamutant. Information from the interface I is included into a driver template, the driver template is then compiled, as is the metamutant, and both driver and metamutant are linked with the metaprocedure library. Lastly the driver is run; running the driver causes the metamutant to be invoked with original program semantics.

The metaprocedure library contains the `BinOp_`, `UO_` `OI_`, `LA_`, and casting routines described in Chapter II. The bulk of this approximately 2,900 statement (5,100 line) library is occupied by the binary operation `BinOp_` routine. The `BinOp_` routine is primarily a nested case construct with approximately $22 \times 8 \times 8$ cases. (The number of C binary operators times the product of the number of computational C data types.)

The `Run Original` tool is a combination of approximately 150 C source statements and approximately 50 UNIX C-shell script lines.

The command "`ro`" invokes the `Run Original` tool.

Analyze Tool

The `Analyze` tool is used to perform mutation analysis. The `Analyze` tool manages and directs the execution of mutants against test cases. The analysis results table *ART* is updated as execution of mutants proceeds.

Much like `Run Original`, every time the `Analyze` tool is used with a new program *P* or a new interface *I*, it creates a "driver" routine capable of invoking the metamutant. Information from the interface *I* is included into a driver template, the driver template is then compiled and both driver and metamutant (the metamutant having already been compiled by the `Run Original` tool) are linked with the metaprocedure library.

When the driver is run, mutants are selected serially from the mutant descriptor list. If that mutant's entry in the analysis results table *ART* is not "`L`", for live, then the next mutant in the list is selected for processing. If that mutant's entry in the *ART* is "`L`", then the metamutant is invoked. The global change point and alternative values are set so that the metamutant will exhibit the semantics for that mutant. The mutant is allowed to execute on the first test case and, on completion, the mutant's output is compared to the expected output. If the mutant's output is different, the mutant is marked killed (a "`D`" entry in the *ART*) and the next mutant in the list is selected for processing. If the mutant's output is not different, the next test case is applied, and so on.

Although this sounds straightforward, it is important to note that once the `Analyze` program begins processing mutants, it must not come to a halt until all of the relevant mutants have had an opportunity to run. This means that no matter what destructive execution behavior a mutant manifests, the driver routine must be able protect itself from this destructive behavior and continue processing.

It is not blatant execution violations, such as zero divide errors, that are the most dangerous to the `Analyze` driver; these can be caught using the operating system's exception handlers. It is insidious memory violations that create the greatest difficulties. It is relatively common for a mutant's mutation to cause an array subscript to go wild or a pointer take on an unwanted value. If the driver and mutant share the same address space, part of the driver's data structures may unknowingly become corrupted causing unpredictable side-effects. At best, the driver aborts and the user is alerted to the problem. At worst, the bookkeeping data structures, like the *ART*, are affected and erroneous mutation adequacy scores get reported.

One option is to run the mutants in their own separate address spaces. Each time a mutant needs to be run, a process is forked and the metamutant runs in that child process. This was demonstrated to work in an earlier version of `TUMS`. Unfortunately forking is computationally expensive.

Our current solution utilizes two coroutine-like concurrent processes, two shared regions of memory, three semaphores to synchronize access to the shared memory regions, and the fact that UNIX (and most other operating systems) stores executable code in a write-protected memory segment. (If executable code is not stored in a write-protected memory segment, then this scheme will not work and the one-process-per-mutant strategy must be used despite its overhead.)

Soon after startup, the `Analyze` tool splits into two concurrent processes. The parent process executes the code of `Analyze.c`: we call it the *driver*. The child process executes the code of `AnalyzePair.c`: we call it the *attendant*. Appendix C contains the source code and related header file for these two programs.

The driver is responsible for determining which mutants are to be run and recording mutant mortality in the analysis results table. The driver is also responsible for obtaining the mutant descriptors from D and test cases from TS.

The attendant is responsible for invoking and monitoring mutants. The mutants run in the attendant's address space, consequently the driver remains unaffected by any mutant behavior.

Communication between the driver and attendant is through the two shared memory segments. One shared memory segment, the Pair-buffer, is attached to the driver as read-write memory and is attached to the attendant as read-only memory. The other shared memory segment, the Results-buffer, is attached to the driver as read-only memory and is attached to the attendant as read-write memory. Access to the shared memory segments is synchronized through the semaphores.

The driver requests that the attendant run a particular mutant on a particular test case by depositing the mutant descriptor and test case information into the Pair-buffer. The driver then blocks awaiting information to be placed in the Results-buffer.

When the attendant unblocks on the Pair-buffer, it uses the mutant descriptor information contained therein to instantiate the metamutant. Test case information is copied over into the Results-buffer where it can freely be manipulated by the mutant. The attendant offers a rampaging mutant very little opportunity to do it harm since the attendant has so very little read-write memory. The attendant regains control of its address space when an exception is detected. The memory used by the signal handlers is not accessible to the attendant or the mutants. Note that since the Pair-buffer is in read-only memory, any attempt by a mutant to destroy that information will result in a segmentation violation that is caught by the attendant and causes the attendant process to reset itself. The one vulnerable spot, or *Achilles' heel*, of the attendant is the memory location it uses to point to the shared memory buffers. It is extremely unlikely that a mutant would inadvertently write to this spot—even so, it is a possibility. Since the attendant must use at least one read-writable storage location with which to attach a shared memory segment, this vulnerability cannot be eliminated.

When the mutant has terminated and the attendant is back in control, the attendant determines if the mutant was killed by using the test case expected results information stored safely in the Pair-buffer to compare against the mutant output. The mutant's status is then placed in the Results-buffer and the lock on that memory is released. The driver, now unblocked, records the mutant mortality information stored in the Results-buffer in the analysis results table. The process then repeats itself until the driver determines there is no more work to do. It then cancels the child process and terminates itself.

The `Analyze` tool is a combination of approximately 325 C source statements and approximately 50 UNIX C-shell script lines.

The command "`an`" invokes the `Analyze` tool.

Mutation Statistics Tool

The `Mutation Statistics` tool is responsible for analyzing entries in the analysis results table *ART* and reporting various statistics, in particular the *mutation adequacy score*. In the `TUMS` prototype this tool reports only one statistic, the mutation adequacy score, and is implemented using 23 UNIX C-shell script commands.

The command "`ms`" invokes the `Mutation Statistics` tool.

Using the System

Although ease of use was not an overriding design factor, the `TUMS` mutation analysis system is almost as easy to use as the command-line version of `Mothra`. The greatest impediment to its general use in software testing is the lack of a "`decode`" tool, that is, a tool that will allow the user to see the mutations or individual mutants in source form.

Assuming the program under test is stored in the file "`sumsqrt.c`" and the test data to be mutation analyzed is stored as "`td.dat`", the `TUMS` command sequence shown in Table IX would be used to determine the mutation adequacy score of "`td.dat`".

Table IX. `TUMS` command sequence for performing mutation analysis.

Step	UNIX command	Description
#1	`mn sumsqrt.c`	Make the program's neighborhood.
#2a	`gi SUMSQRT td`	Create the interface between the program neighborhood *N* and the test data *D*. In this example, a sub-directory called `SUMSQRT+sumsqrt` will be created with five interface templates files in it.
#2b	`vi` *TEMPLATE*	Edit the template files, as needed.
#3	`ro SUMSQRT td`	Run the original program to (1) see what happens, (2) create the test set *TS*, and (3) initialize the *ART*
#4	`an SUMSQRT td`	Analyze the test set relative to the neighborhood.
#5	`ms SUMSQRT td`	Determine the mutation adequacy score.

CHAPTER IV

EXPERIMENTS AND EMPIRICAL RESULTS

This chapter presents several empirical results that relate the performance of mutation analysis using the `MSG` method to previous and hypothesized methods. The first section of this chapter describes the suite of specimen programs used in the studies and the platform environment. Section two compares the performance of `TUMS` vis-à-vis `Mothra`. A hypothetical "ideal" mutation analysis system is proposed in section three and used to provide a baseline in a statistical bounds of performance study. The aggregate performance of schema-based mutation analysis systems is explored in section four. This chapter concludes that mutation analysis using the new `MSG` method is significantly faster than using the conventional method, with speed-ups as high as an order-of-magnitude observed.

The Specimen Programs and Platform Environment

A suite of eight programs was chosen to use as experiment specimens in the various studies. They are listed alphabetically in Table X. These programs represent a mix of data types (scalar and array, floating-point and integer), a range of program neighborhood sizes, and a variety of control path complexities. This mix of characteristics is listed in Table XI.

Table X. The specimen program suite.

Program	Short Description
CHI	Apply Chi-square test to N pseudo-random numbers (`float` arrays).
CPRIMES	Count prime numbers in range $1 \ldots N$, using `double` arithmetic.
FIND	Find the Fth largest element in an array $A[1 \ldots N]$.
ICHI	Apply Chi-square test to N pseudo-random numbers (`int` arrays).
ICPRIMES	Count prime numbers in range $1 \ldots N$, using `int` arithmetic.
LFIBO	Large (100 digit) precision Nth Fibonacci number function.
SUMSQRT	Calculates the sum of the square roots of $1 \ldots N$.
TRITYP	Categorizes triangles given the lengths of their sides.

Table XI. Specimen program characteristics.

Program	Data Types				#Mutants in C	#Mutants in Fortran	Cyclomatic Complexity
	Array	integer	float	double			
CHI	√	√	√		2423	2173	4
CPRIMES		√		√	636	540	6
FIND	√	√			1067	1022	8
ICHI	√	√			2307	2091	4
ICPRIMES		√			552	405	6
LFIBO	√	√			4075	3713	13
SUMSQRT			√		707	590	4
TRITYP		√			1403	951	18

The `FIND` program, developed and analyzed by Hoare [59, 60], accepts as input an N element array A and an integer index F and finds the Fth largest element of the array. It rearranges the array in such a way that this element is placed in `A[F]`; furthermore, all elements with subscripts lower than F have values less than or equal to `A[F]` and all elements with subscripts greater than F have values that are greater than or equal to `A[F]`.

The `TRITYP` program, widely used as an example because of its easy description yet relatively high cyclomatic complexity[12], categorizes triangles as either equilateral, isosceles, scalene or illegal, given the lengths of their sides, represented as three integer input values. The programs `FIND` and `TRITYP` are considered "classics" in the program testing literature.

The `LFIBO` program accepts as input an integer N and calculates the Nth Fibonacci number. The output Fibonacci number is represented as an array of digits. This program exhibits high cyclomatic complexity, has a large program neighborhood and is the largest program in the suite.

Typically programs selected in testing studies have short run times. The programs above possess that characteristic. However, for the bounds of performance study described in section three, programs that could, given the right input, exhibit a wide range of execution times were needed. The following programs satisfy this requirement.

---

[12]McCabe's *cyclomatic complexity* [61] is a widely used software metric based on the control flow properties of a program. For structured programs, such as those in Table X, the metric can be calculated by adding one to the number of distinct predicates in the program.

The CHI program generates N pseudo-random numbers in the range $0 \ldots 99$ using the linear congruential method. It then applies the chi-square statistical test on the sequence of numbers produced by this scheme to see how "random" the distribution is. The ICHI program is an all integer variant of CHI.

The ICPRIMES program uses a straightforward but inefficient scheme to count the numbers in the range $1 \ldots N$ that are prime. The CPRIMES program is a variant of ICPRIMES that employs double precision floating point arithmetic.

The SUMSQRT program uses single precision floating point arithmetic in calculating the sum of the square roots of the integers $1 \ldots N$. The individual square roots are determined using Newton's method. Although compact, this program contains features representative of many programs, such as nested WHILE loops and an IF statement.

An ANSI C language implementation (*.c) and a Fortran language implementation (*.f) of each program was prepared. Great care was exercised to make the C and Fortran versions of the programs as alike as possible, that is, to use comparable control flow constructs, similar variable names and constants, and formatting. The cyclomatic complexity of the C and Fortran versions is identical. The source code for these implementations is found in Appendix D.

All experiments were run on Sun 4/25 (ELC) workstations[13]. The characteristics of this platform are given in Figure 19. All metamutant and TUMS programs were compiled using version 2.5.8 of the GNU gcc compiler using optimization level one (-O1) and the "-freg-struct-return" code generation option. The Mothra interpreter tool rosetta, consisting of components intdriver version 9.2 and interp version 9.1, was compiled with the standard SunOS 4.1 cc compiler using optimization level two (-O2)[14].

[13]For comparison, the Compaq Deskpro PC with a 486DX/33 processor, although a bit slower in doing floating point arithmetic, has an identical integer SPECmark speed rating of 18.2.

[14]This produced the fastest running version of rosetta. Curiously, higher optimization levels slowed rosetta down. Also, the fastest gcc compiled version of rosetta ran approximately 15% slower than the cc compiled version used in the studies.

§ **Hardware**

- Sun SparcStation ELC (Sun 4/25)
- RISC architecture
- SPARC CPU Model #FJMB86903
- Weitek 3170-based 33MHz FPU
- 16 Megabytes of RAM

§ **Operating System**

- SunOS 4.1.1 operating system

§ **Performance**

- 10460 Whetstone KIPS (single precision)
- 18.2 SPECint-92 17.9 SPECfp-92

Figure 19. Platform characteristics.

The `TIMER` program listed in Appendix E was used to measure program execution times. This "stopwatch" program was needed since the times measured by using the Unix C-shell built-in `time` command sometimes failed to add the CPU time spent in child processes to the total time. The `TIMER` program uses two alternate Unix system timing facilities in measuring program execution. These two values were used as cross-checks; timings for programs where these two values did not correspond were not used. Program execution times are CPU headway times, not elapsed wall clock time, and are the sum of the user and system times. Unless otherwise noted, execution times reported in this dissertation are in milliseconds (ms).

TUMS versus Mothra Performance Study

This research was prompted by dissatisfaction with the computational expense of performing mutation analysis using conventional interpretive methods. To investigate whether a schema-based mutation analysis system outperforms an interpreter-based system, the `TUMS` system was benchmarked against the `Mothra` system. `Mothra` is the most comprehensive of the conventional interpretive mutation analysis systems [33, 34].

A complicating factor in the comparison of `TUMS` to `Mothra` is that `TUMS` processes C language programs whereas `Mothra` processes Fortran language programs. It is necessary to have program neighborhoods of similar constituency and size for the comparison of mutation analysis times to be meaningful. Unfortunately few C and Fortran mutagens produce corresponding types and numbers of mutants. This is true even of mutagens that superficially are identical. For example, the C *comparable operator replacement* (`Ocor`) mutagen and the Fortran *arithmetic operator replacement* (`AOR`) mutagen were designed to induce the same mutations. However C has far more binary operators than Fortran resulting in `Ocor` program neighborhoods containing many mutants that have no analog in `AOR` program neighborhoods. Similarly, the C `SSDL` and Fortran `SDL` are both *statement deletion* mutagens but, because C permits recursive statement definitions and compound statements, the C `SSDL` program neighborhood will be larger.

Table XII lists the C mutagens in `TUMS` that have closely corresponding Fortran counterparts in `Mothra`. The left column identifies the C mutagen and the right column lists the set of one or more Fortran mutagens that produces approximately the same types and numbers of mutants as the C mutagen.

Table XII. Closely corresponding C and Fortran mutagens.

C Mutagen		Corresponding Fortran Mutagen(s)	
Name	Description	Name	Description
Vssr	Scalar for Scalar Replacement (VGSR + VLSR)	AAR ASR SAR SVR	Array Ref. for Array Ref. Array Ref. for Scalar Variable Scalar Variable for Array Ref. Scalar Variable for Scalar Variable
Vcsr	Constant for Scalar Replacement (VGCR + VLCR)	CAR CSR	Constant for Array Ref. Constant for Scalar
Cscr	Scalar for Constant Replacement (CGSR + CLSR)	ACR SCR	Array Ref. for Constant Scalar for Constant
Cccr	Constant for Constant Replacement (CGCR + CLCR)	SRC	Source Constant
SGLR	`goto` Label Replacement	GLR	`GOTO` Label Replacement

The results of applying the mutagens from Table XII to the specimen programs is shown in Table XIII. Although the C and Fortran program neighborhoods are close in size and composition, there are still cases where the neighborhoods do not correspond exactly. One reason C and Fortran program neighborhoods may not correspond exactly is because the definitions of most Fortran mutagens contain *restrictions* [34]: simple rules that seek to inhibit the generation of some mutants that are either equivalent to the original program or to other mutants.[15]

[15]Note: Using such ad-hoc restrictions to reduce the amount of redundant execution is problematical. Simple restrictions have severe limits in their ability to eliminate redundant mutants and, worse, react in unplanned ways to occasionally cause some necessary mutants from being produced. For example, unless the `CRP` mutagen is specified along with the `SRC` mutagen, certain mutants that should be produced are not. Program analysis techniques, borrowed from compiler technology, might prove useful in identifying and discarding redundant mutations.

Table XIII. Mutants produced by language per mutagen.

Program	Language	Mutants Produced per Mutagen Vssr	Vcsr	Cscr	Cccr	SGLR	Combined Total
CHI	C	689	300	392	252	0	1633
	Fortran	638	280	463	251	0	1632
CPRIMES	C	120	60	66	33	12	291
	Fortran	115	60	61	38	12	286
FIND	C	480	32	55	0	84	651
	Fortran	467	68	54	5	84	678
ICHI	C	689	240	392	196	0	1517
	Fortran	638	238	463	214	0	1553
ICPRIMES	C	80	48	60	36	12	236
	Fortran	76	41	55	13	12	197
LFIBO	C	1680	287	882	252	30	3145
	Fortran	1676	283	869	138	22	3030
SUMSQRT	C	180	72	63	27	0	342
	Fortran	178	64	59	21	2	324
TRITYP	C	141	175	80	80	0	476
	Fortran	141	160	71	46	0	418

For instance, in applying the C `Vssr` mutagen, if given the expression "`X = B`" one of the mutations the `TUMS` system will produce is "`X = X`" by replacing the scalar variable `B` by the scalar variable `X`.[16] The `Mothra` system will not produce such a mutation since the corresponding Fortran `SVR` mutagen has the restriction: *"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 would be equivalent to an* `SDL` *mutant; e.g.,* `X = X` *is equivalent to* `CONTINUE`.*)*". Similarly, given the expression "`X = 5`", the C `Cscr` mutagen will require the `TUMS` system to produce the mutation "`X = X`" whereas `Mothra` will not produce the corresponding mutation because of a restriction placed on the Fortran `SCR` mutagen. The C `Vcsr` mutagen will cause `TUMS` to mutate the expression "`A + B`" to "`A + 0`" whereas no corresponding mutation will occur in the `Mothra` system because of

[16] Note that in C such a mutation should not be inhibited since the assignment expression might appear inside a statement, such as "`if (X = B) ...`"; the mutation "`if (X = X) ...`" *must* be generated.

restrictions on the Fortran `CSR` mutagen. The greatest disparity exists between the C `Cccr` mutagen and the Fortran `SRC` mutagen. The two restrictions on the Fortran `SRC` mutagen that cause problems are: "*An integer constant is not replaced by another integer constant whose value differs by plus or minus one (equivalent to a* `CRP` *mutant).*" and "*Constant replacement is not performed when so doing would create one of the following mutants:* `X + 0, 0 + X, X - 0, X * 1, 1 * X, X / 1, X ** 1` *(all are equivalent to* `AOR` *mutants), or* `X / 0` *(equivalent to a* `SAN` *mutant).*" These two restrictions often result in only half as many Fortran `SRC` mutants as C `Cccr` mutants being generated. Finally, the C `SGLR` mutagen may produce more mutations than the Fortran `GLR` mutagen depending on the way loops and `if` statements are nested within the program.

Another complication arising from the fact that `TUMS` and `Mothra` process different source languages stems from minor differences in the C and Fortran implementations of the same program. For example, arrays in C are zero-origin indexed and are declared by specifying the extent (the number of elements) of the array. Thus an integer array `A` whose subscripts range $0 \ldots 99$ would be declared as "`int A[100];`". By default, Fortran arrays are one-origin indexed. Thus to declare a corresponding array, "`INTEGER A(0:99)`" would be written. This difference in declarations leads to a difference in the sets of constants used in the *Constant for Scalar* and *Constant for Constant* replacements. The C program contains the constant `100` whereas the Fortran program contains the two constants `0` and `99`. Differences in control structures between the two languages also pose problems. Although the semantics of the C `for` loop are similar to the Fortran `DO` loop, they differ greatly in their syntactic form and consequently they mutate differently. Also, the C language possesses a `while` loop that has no direct analog in Fortran. Thus C program and Fortran program neighborhoods may contain some incomparable mutants.

Table XIV shows which mutagens were applied to each specimen program to construct the C program neighborhoods used in the `TUMS` to `Mothra` comparisons. The `TUMS` system was designed to allow individual mutants to be disabled, that is, to be marked in such a way that they are excluded from the program neighborhood and ignored during the mutation analysis. Where necessary, the C program neighborhoods were manually examined and, wherever a C mutant that did not have a corresponding Fortran mutant was found, those C mutants were disabled. This resulted in equal sized neighborhoods being compared. The exception is the `LFIBO` program whose C program neighborhood is a negligible 0.3% larger; the size of the neighborhood made identifying all the incomparable mutants impractical. Column 9 of Table XIV shows how many mutants were disabled and thus excluded from the C program neighborhoods.

Table XIV. Mutagens used in program neighborhood construction.

	Mutagens Used					Number of Mutants		
Program	`Vssr`	`Vcsr`	`Cscr`	`Cccr`	`SGLR`	Initial	Disabled	Used
CHI				√		252	1	251
CPRIMES		√				60	0	60
FIND	√		√		√	619	14	605
ICHI		√				240	2	238
ICPRIMES	√		√		√	152	9	143
LFIBO	√	√			√	1997	10	1987
SUMSQRT			√	√		90	10	80
TRITYP	√	√	√	√		476	58	418

The results of the `TUMS` to `Mothra` comparison are summarized in Tables XV and XVI. These results were obtained using the following procedure. For each specimen program, a test set sufficient to kill at least 70% of mutants was created. These test sets, A through H, are listed in Appendix F. For each specimen program, the corresponding test set was mutation analyzed by the `TUMS` system. To assure representative times, the mutation analysis was performed three times; the analysis with the median total time was used. Similarly each test set was mutation analyzed by the `Mothra` system; these analyses were done three times and the analysis with the median total time was used.

Table XV. TUMS **speed-up** vis-à-vis Mothra.

Program	Test* Set	System	Number Mutants	Setup Time (in *ms*)	Run Original Time (in *ms*)	Run Mutants Time (in *ms*)	Total Analysis Time (in *ms*)
CHI	A	TUMS	251	32860	1550	303520	337930
		Mothra	251	550	52680	10746480	10799710
Speed-up →					**34.0**	**35.4**	**32.0**
CPRIMES	B	TUMS	60	28630	1320	531570	561520
		Mothra	60	500	50050	10602180	10652730
Speed-up →					**37.9**	**19.4**	**19.0**
FIND	C	TUMS	605	33690	750	5240	39680
		Mothra	605	730	630	88360	89720
Speed-up →					**0.8**	**16.9**	**2.3**
ICHI	D	TUMS	238	32880	1500	386790	421170
		Mothra	238	510	52720	11471340	11524570
Speed-up →					**35.1**	**29.7**	**27.4**
ICPRIMES	E	TUMS	143	28790	680	87040	116510
		Mothra	143	630	6160	2338070	2344860
Speed-up →					**9.1**	**26.9**	**20.1**
LFIBO	F	TUMS	1987	54800	1490	53430	109720
		Mothra	1981	1060	8380	1423460	1432900
Speed-up →					**5.6**	**26.6**	**13.1**
SUMSQRT	G	TUMS	80	29560	1470	53820	84850
		Mothra	80	490	62320	1339560	1402370
Speed-up →					**42.4**	**24.9**	**16.5**
TRITYP	H	TUMS	418	34150	780	5110	40040
		Mothra	418	650	1810	121660	124120
Speed-up →					**2.3**	**23.8**	**3.1**

*See Appendix F for test set contents.

Table XVI. Test sets and mutation analysis scores.

Program	Test* Set	#Test Cases	`TUMS` #Mutants	`TUMS` Killed	`TUMS` *MS*‡	`Mothra` #Mutants	`Mothra` Killed	`Mothra` *MS*‡
CHI	A	1	251	229	91	251	203	81
CPRIMES	B	1	60	60	100	60	60	100
FIND	C	7	605	590	98	605	590	98
ICHI	D	1	238	224	94	238	221	93
ICPRIMES	E	1	143	136	95	143	136	95
LFIBO	F	7	1987	1865	94	1981	1880	95
SUMSQRT	G	3	80	61	76	80	61	76
TRITYP	H	34	418	409	98	418	409	98

*See Appendix F for test set contents.

‡Calculated without considering equivalent mutants; adjusted *MS* would be higher.

The *"Setup Time"* column of Table XV shows the one-time costs of preparing a program for mutation analysis. In `TUMS`, this setup time is the sum of the time needed to create the metatmutant and mutant descriptor file (i.e., the run-time of the `MakeNeighborhood` tool) and the compilation times for the metamutant and its drivers. In `Mothra`, this setup time is sum of the time needed to translate the program into intermediate code and the time needed to generate the mutant descriptor records; that is, the combined run-times of the `parse` and `mutmake` tools. The *"Run Original Time"* column lists the execution times under each system for the original program to process the test set. This, in essence, is the time needed to test the original program since we do not mutation analyze a test set if the original program manifests a failure in processing that test set. The *"Run Mutants Time"* column records how long it took to run the series of mutants against the test set under `TUMS` and `Mothra`. Representing the sum of all the mutant program execution times, this is the costliest step in mutation analysis. The *"Total Analysis Time"* column contains the overall time needed to perform a mutation analysis. It is the most telling measure of performance and is the sum total of the three previous columns.

The *speed-up* of TUMS vis-à-vis Mothra is computed by dividing the Mothra execution times by the TUMS execution times. Examining the last column of Table XV, it can be seen that the amount of overall speed-up is dependent on the specimen program. The overall speed-ups range from a low of 2.3 to a high of 32.0.

The FIND program, with a speed-up of 2.3, and the TRITYP program, with a speed-up of 3.1, show the least overall speed-up. Although a significant improvement over Mothra, these speed-ups do not match the improvements seen with the other programs. Examining the mutation analysis times in the next to last column, FIND and TRITYP have speed-ups of 16.9 and 23.8, respectively. These "classics" of the software testing literature perform very little computation and thus have very short execution times. Consequently the larger setup time required under TUMS overshadows the very short analysis times. In general, we would expect other programs with small computational demands to exhibit similar results. Of course, such quick running programs by their very nature are quickly analyzed by either TUMS or Mothra.

For the rest of the specimen programs, the TUMS system exhibits order-of-magnitude improvements in performance relative to Mothra. Generally the greater the computational demands of the specimen program, the greater the improvement. Thus the LFIBO program, with the smallest computational demands in this group, has a speed-up of 13.1 whereas the CHI program, which does more computation, has a speed-up of 32.0. For the CHI program, it takes TUMS approximately $5\frac{1}{2}$ *minutes* to perform the mutation analysis. In contrast, it takes Mothra 3 *hours* of computer time! In general, we would expect any other programs with medium-to-large computational demands to exhibit similar results.

These benchmarks empirically establish that TUMS greatly outperforms Mothra.

Bounds of Performance

It is possible to define a hypothetical *"ideal"* mutation analysis system that executes with *best possible* speed. Using the performance of such an "ideal" system as a baseline, the performance bounds of other mutation analysis systems can be determined.

For ease of reference, we shall refer to this hypothetical "ideal" system as `IDEAL`. To make comparisons meaningful, we require the `IDEAL` mutation analysis system to operate on the same standard neighborhoods (as defined in Chapter I) as `TUMS` and `Mothra`. Consequently, given a test set, the `IDEAL` system must execute the original program and all live mutants against that test set. The `IDEAL` system is allowed to assume that executable versions of the program and mutants are "magically" available when needed; we allow our hypothetical system to ignore the considerable costs of generating, compiling, linking, and storing these executables. However we do require these executables to resemble what would be produced by running a real compiler[17] and hence execute at rates consistent with real executables. No overhead in switching between program variants is assessed against the `IDEAL` system. We posit that no real mutation analysis system, encumbered by costs that our hypothetical `IDEAL` system is allowed to ignore, can execute faster. Thus the hypothesized execution times of the `IDEAL` system can be used as a lower bound—a baseline against which other systems can be compared.

All results in this chapter are *conservative* in the classic sense. That is, in any comparisons between `TUMS` and other systems, the benefit of the doubt is accorded the other system. Thus, for example, in obtaining run-time estimates of the `IDEAL` system, we use the lowest values possible for our estimates; we shall in fact identify these `IDEAL` values as the *"conservative low"* times.

The performance bounds of four execution systems were studied: `FastTwin TUMS`, `SlowTwin TUMS`, `CX`, and `Mothra`. As previously described in Chapter II, in `TUMS` when metamutants are generated using the *twinning* strategy each statement actually appears twice albeit in two forms: a *slow* fully metamutated form and a *fast* minimally modified form. When the metamutant runs representing the functionality of a mutant, the *slow twin* version of a statement is executed only if that statement contains the mutation. Otherwise the *fast twin* version of the statement is run. When the twinning strategy is not used in

---

[17] In this case, the GNU `gcc` compiler as described on page 65.

generating the metamutant, each statement appears only in slow fully metamutated form. By FastTwin TUMS, we shall refer to execution of a metamutant where both fast and slow forms of a statement are present.[18] Note that when such metamutants execute as the original program (i.e., no mutation present) only fast form statements get executed. By SlowTwin TUMS, we shall refer to execution of a metamutant where all statements are in slow form.

CX is a C language interpreter drawn from the UPS debugging system [62]. It is considered to be a very efficient interpreter whereas the efficiency of the internal Mothra interpreter (rosetta) is unknown. Thus results from using the fast CX interpreter might be generally representative of any performance-minded interpreter-based execution system.

For the five specimen programs CHI, CPRIMES, ICHI, ICPRIMES, and SUMSQRT, the time needed to execute the original program for seven different test cases using FastTwin TUMS, SlowTwin TUMS, CX[19], and Mothra was plotted against the corresponding time it would take the IDEAL system to complete the same work. These plots are shown in Figures 20 through 24. To determine the time it would take the IDEAL system to complete an execution, each specimen program was compiled using the GNU gcc compiler, linked to the same TUMS driver harness program that invokes a compiled metamutant, and then run seven times. Of these seven runs, the *lowest* observed time was recorded as the "conservative low" for each workload and used as the baseline value. The test cases for each specimen program were tailored so that the baseline (workload) execution times would be approximately 1000, 3000, 5000, 7500, 10000, 60000, and 100000 milliseconds long. These test cases, contained in test sets I through M, are listed in Appendix F.

The data values used to prepare the plots in Figures 20 through 24 are detailed in Appendix G. Note that seven timings were collected for each test case to assure representative results and permit further statistical analysis.

---

[18]This is the default type of metamutant generated by TUMS .

[19]Because the CX interpreter is unable to pass float * parameters, it was unable to execute the CHI and SUMSQRT programs.

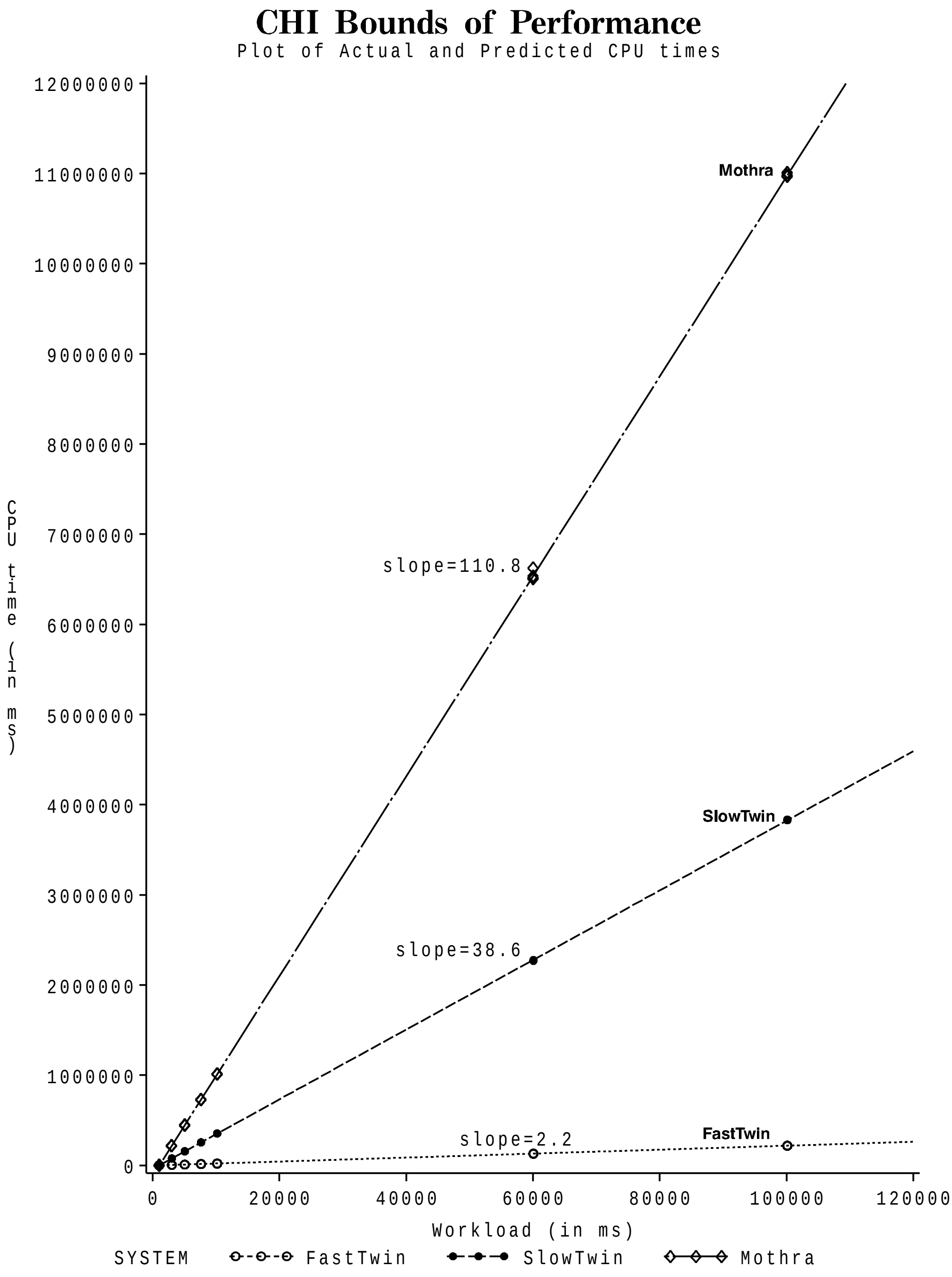

Figure 20. CHI bounds of performance.

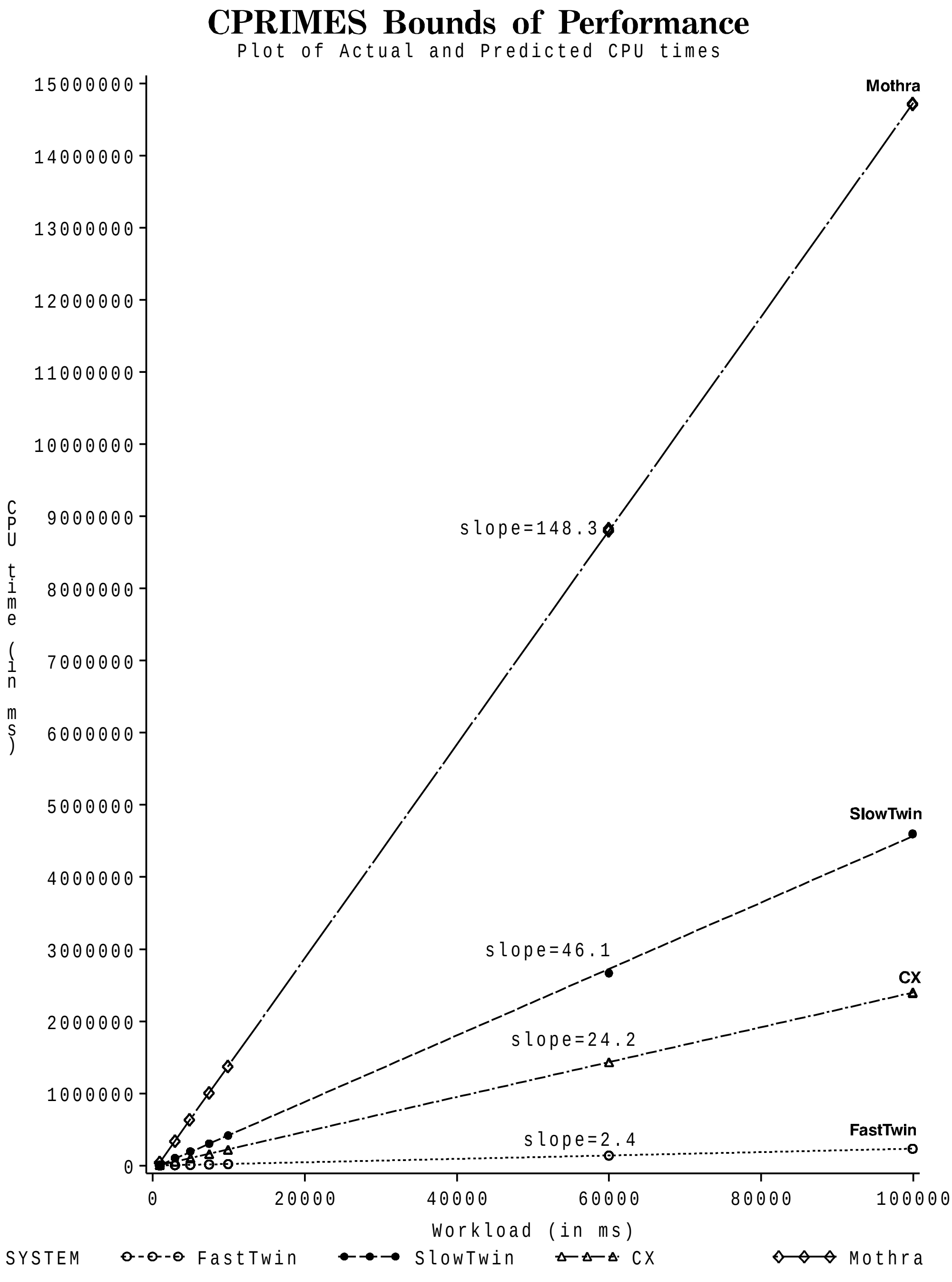

Figure 21. CPRIMES bounds of performance.

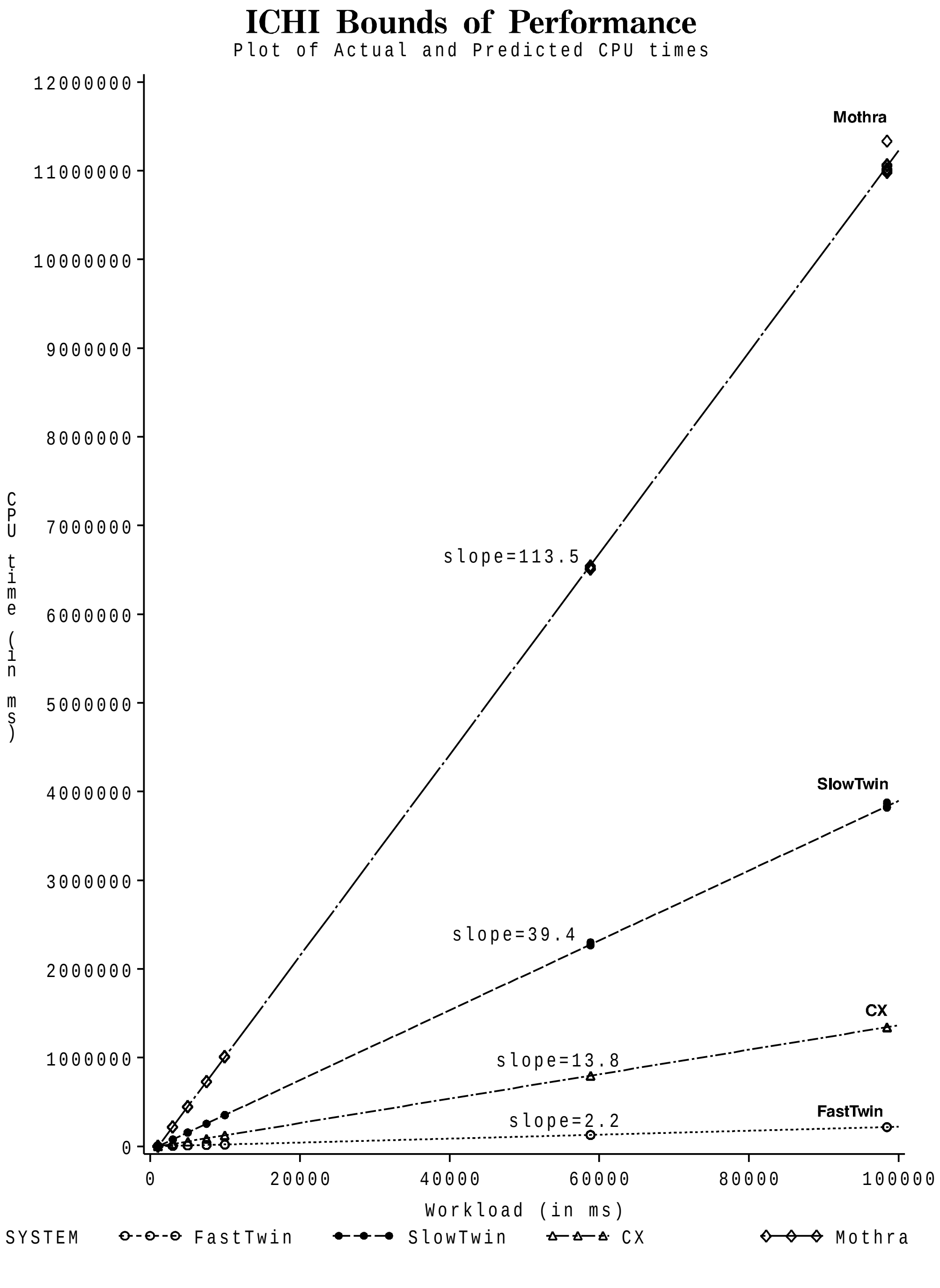

Figure 22. ICHI bounds of performance.

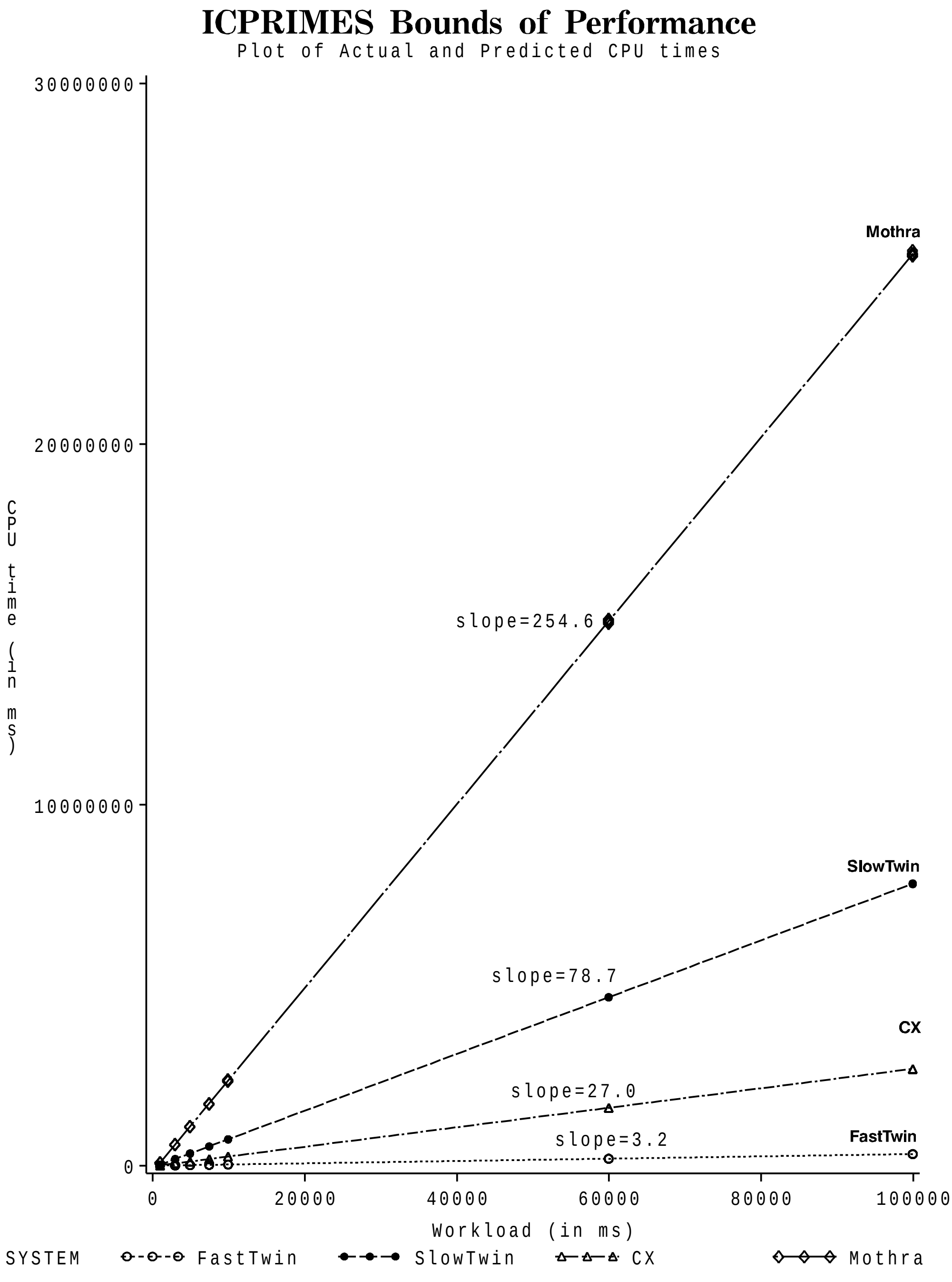

Figure 23. `ICPRIMES` bounds of performance.

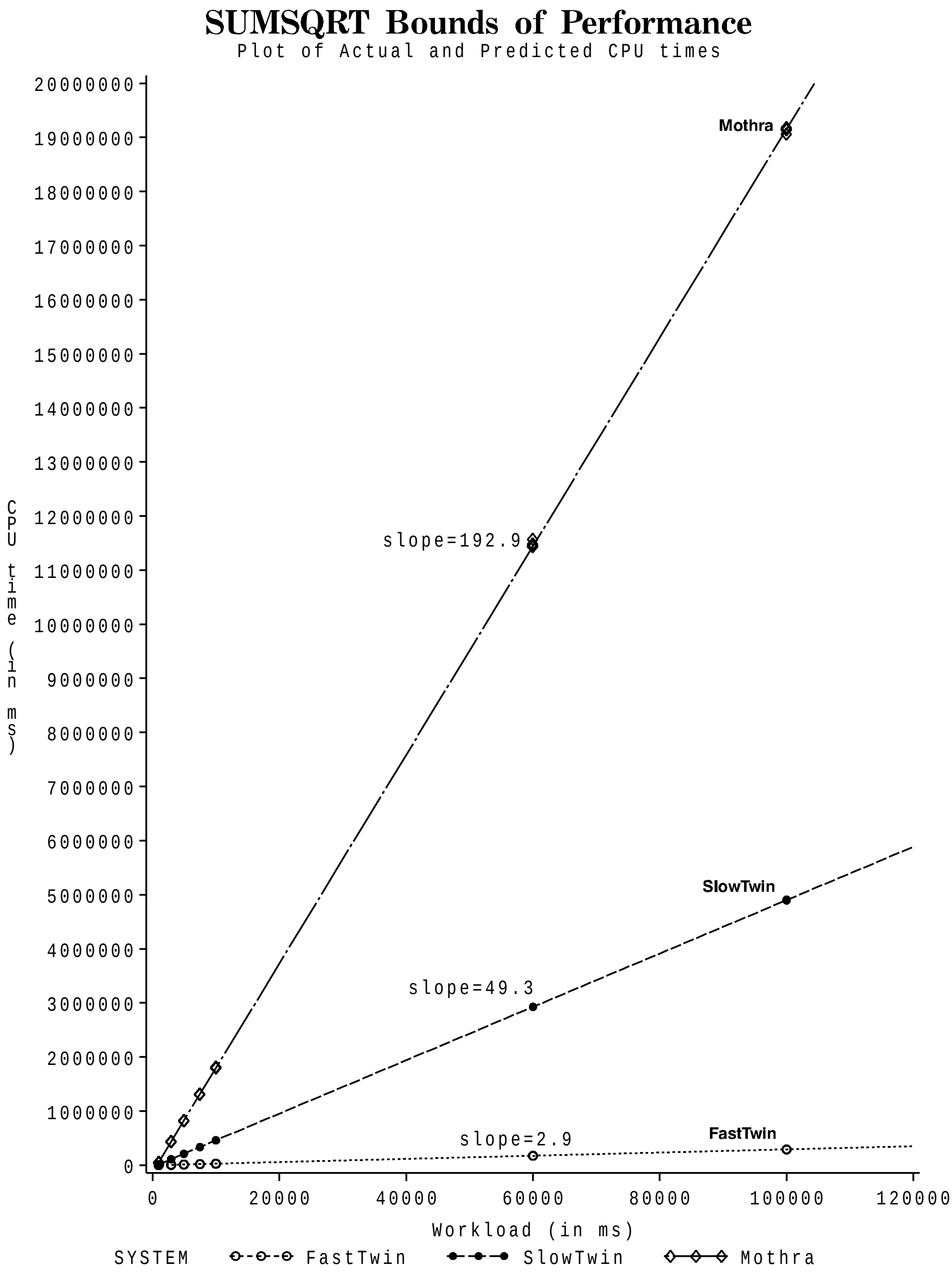

Figure 24. `SUMSQRT` bounds of performance.

From this data, least squares estimates for the parameters in the model

$$y = \beta_0 \ + \ \beta_1 x \ + \ \epsilon,$$

where y denotes CPU time, x the workload, and ϵ is a random error, were obtained for each of the execution systems, `FastTwin TUMS`, `SlowTwin TUMS`, `CX`, and `Mothra`. The corresponding regression lines are shown on the plots in Figures 20 through 24. The β_1 parameter, or *slope*, of each regression line is also noted in the plots. For each model, the coefficient of determination was calculated. The *coefficient of determination*, or R^2 value, for a model is defined as the proportion of the variability in the predicted, or dependent, variable (in this case, CPU time) that can be accounted for by the explanatory variable (in this case, workload) of the model [63]. The R^2 value provides a summary measure of how well the regression line fits the data. For example, an R^2 value of 0.99 for a particular regression model means that the explanatory variable explains 99% of the variability in the y values. All models had $R^2 > 0.99$. Also the observed significance level, or ρ value, of each β parameter was calculated and found to be less than or equal to .0001 implying these parameter estimates are statistically significant. Calculations were done using the SAS statistical package [64]; the results are reproduced in Appendix G.

Using the slopes of the regression lines to characterize the relative execution speeds of the four systems, the results of this study of performance bounds are summarized in Table XVII.

Table XVII. Bounds of performance summary.

Program	Slopes			
	`TUMS`		`CX`	`Mothra`
	FastTwin	SlowTwin	interpreter	interpreter
CHI	2.2	38.6	n/a	110.8
CPRIMES	2.4	46.1	24.2	148.3
ICHI	2.2	39.4	13.8	113.5
ICPRIMES	3.2	78.7	27.0	254.6
SUMSQRT	2.9	49.3	n/a	192.9

The CX interpreter executes between 13.8 and 27.0 times slower than the baseline IDEAL system.[20] This is so much faster than the Mothra system, which performs between 110.8 and 254.6 times slower than the baseline, that it seriously calls into question the use of Mothra, performance-wise, as a representative interpreter-based mutation analysis system.

The *best-case* behavior of TUMS is demonstrated by the performance of FastTwin TUMS. We see that FastTwin TUMS is only between 2.2 and 3.2 times slower than the baseline IDEAL system. The *worst-case* behavior of TUMS is demonstrated by the performance of SlowTwin TUMS. We see that SlowTwin TUMS is somewhere between 38.6 and 78.7 times slower than the baseline IDEAL system. When executing mutants, TUMS will execute within the performance envelope described by these upper and lower bounds.

Aggregate Performance

If we posit that an interpreter-based mutation analysis system could be built that executes at speeds comparable to the CX interpreter, then the "aggregate" performance behavior of TUMS becomes important since the performance of CX lies within the TUMS upper and lower performance bounds.

What is *aggregate* performance? Recall that the total execution cost incurred during mutation analysis is the sum of the times it takes to execute the original program and each extant live mutant program against each test case. In TUMS, a single metamutant represents the functionality of the original and mutant programs. When a metamutant[21] executes acting like the original program, it runs at the upper performance bound (i.e., with the least slowdown relative to the IDEAL baseline) since no mutated statements need to be executed. When acting as a mutant program, the degree to which the metamutant runs slower is proportional to the frequency with which the mutated statement is executed. If the

[20] Because the CX interpreter is unable to pass `float *` parameters, it was unable to execute the CHI and SUMSQRT programs. We anticipate the CHI program would have executed only slightly faster than the ICHI program and the SUMSQRT program would likely have had a performance slope lower than ICPRIMES and thus would not have contributed to establishing a performance range upper bound.

[21] We refer here to a default FastTwin TUMS metamutant containing both *fast* and *slow twin* statements.

mutated statement is executed only once, as can easily happen when the mutated statement lies outside of any loops, the metamutant will be performing at close to best-case behavior. If the mutated statement is executed with great frequency, metamutant performance will degrade accordingly. Thus aggregate performance is the weighted average of the varying actual-to-baseline execution ratios encountered during mutation analysis. It is a *weighted* average since the amount of work performed per test case varies from mutant to mutant. The bulk of the mutants do about as much work as the original program. However some mutants quickly terminate execution and, at the other extreme, some may enter a loop that causes an eventual time-out. The actual amount of work performed by each mutant depends on the effect of the mutation and in general cannot be predicted *a priori*.

Since the aggregate performance is a weighted average, for any given mutation analysis task (i.e., program–test set pair) it can be calculated by dividing the total `TUMS` execution time by the total `IDEAL` (baseline) execution time. Obtaining the total `TUMS` time is simple, we perform the mutation analysis and measure the CPU time consumed. Obtaining the total `IDEAL` time is a challenge since the system does not exist!

In the bounds of performance study, only the times needed to run the unmutated original programs were required. The `IDEAL` baseline values were obtained by compiling the relevant specimen program and executing it against different test cases. To determine *total* `IDEAL` time in this fashion, *all* of the mutants would need to be separately generated, compiled, and executed. To do this manually is grossly impractical—the five specimen programs used in the previous section have a combined total of 6625 mutants, with individual neighborhood sizes ranging between 552 and 2423 mutants. Clearly another approach to estimating the `IDEAL` times is needed.

The time required for any program to execute is the sum total of the time spent executing each statement of the program. If a tally of how often each statement was executed is available and if each statement's execution cost is known then the product of these quantities yields the time required to execute the entire program. Since `TUMS` executes each mutant, it was possible to modify the `TUMS` system to produce statement execution frequency profiles for each mutant, that is, tallies of how often each statement was executed per mutant per test case. However identifying a statement's execution cost through direct measurement is impossible. The time required to execute a statement is smaller than the granularity of the computer's clock.

In an attempt to indirectly ascertain the individual statement costs, the original program was run under a variety of workloads (i.e., using different test cases) and the program execution times and the corresponding statement execution tallies were recorded. From this data, for each of the five specimen programs, least squares estimates were obtained for the parameters in the model

$$T = \beta_0 \ + \ \beta_1 S_1 \ + \ \beta_2 S_2 \ + \ldots + \ \beta_n S_n \ + \ \epsilon,$$

where T denotes program execution time, S_1 how often statement 1 executed, S_2 how often statement 2 executed, and so on through statement n of the n-statement program, and ϵ is a random error. The β_1 through β_n parameter values were meant to represent the execution cost of the corresponding statement. However most of the β parameters were estimated as zero. On reflection, it was apparent that some statement costs were being consolidated, or lumped together. For example, all statements in a basic block[22] are executed the same number of times, consequently if S_A was the statement tally for the first statement of the basic block, the execution costs for the entire basic block were consolidated into the β_A value. Removing those S_i's for which the corresponding B_i was zero resulted in a set of reduced estimation equations for each specimen program.

The estimated execution time under the hypothetical `IDEAL` system for each mutant was calculated by plugging statement tally values obtained from the corresponding `TUMS` generated mutant statement execution profile into the appropriate estimation equation.

---

[22] A *basic block* is a program fragment that has only one entry point and whose transfer mechanism between statements is that of proceeding to the next statement [65]. If any statement in the block is executed, all statements in the block are executed.

Because not all of a mutant's statement tallies are used in the reduced estimation equations, some of the initial estimates were clearly incorrect. Upon inspection, this typically occurred when the mutant either terminated execution prematurely or when the mutant entered an infinite loop and was subsequently timed-out. To accommodate cases where the statement execution tallies were atypical, the estimation equations were modified to accept correction factors. The final estimation equations are given in Appendix H for each specimen program.

Completely validating the final estimation equations without a working system is not feasible, but limited *spotchecking* is possible and practical. For each of the five specimen programs, two mutants from the program neighborhood were arbitrarily selected. Each of the two mutants was generated by manually mutating the original program. The mutants were then compiled, executed and their actual execution time recorded. This actual time was compared to the estimated execution time. The results are shown in Table XVIII.

Table XVIII. Spotchecks of estimated hypothetical `IDEAL` times.

Program	Test Set*	Mutant ID	Estimated Time	Actual Time	Difference
CHI	N	#1293	2005	1880	+7%
	N	#1766	2005	2030	-1%
CPRIMES	O	#269	1530	1540	-1%
	O	#393	9251	9260	0%
ICHI	P	#307	1807	1800	0%
	P	#2307	1970	1970	0%
ICPRIMES	Q	#26	1664	1780	-7%
	Q	#535	3978	4080	-3%
SUMSQRT	R	#206	3799	3770	+1%
	R	#707	1130	1130	0%

*See Appendix F for test set contents.

The last column of Table XVIII shows what percent the estimated time is over or under the actual time. Several of the estimates are identical or virtually identical to the actual values. The worst of the estimates are only about 7% off.

To determine the aggregate performance of TUMS for each of the five specimen programs, the complete set of mutagens was used thereby creating standard neighborhoods. The actual total execution time for each mutation analysis using TUMS was recorded. The total IDEAL execution time was calculated by adding together the individual mutant execution estimates. For each specimen program, the total TUMS execution time was divided by the corresponding total IDEAL time to obtain the potential speed-up of the hypothetical IDEAL system vis-à-vis the TUMS system: this value represents the aggregate performance of TUMS. These results are posted in Table XIX.

Table XIX. Potential speed-up of hypothetical IDEAL system vis-à-vis TUMS.

Program	Test Set*	Number Mutants	Mutants Killed	Number Time-outs	Estimated Hypothetical IDEAL Time	TUMS Actual Time	Potential Speed-up
CHI	N	2423	2232	73	4706923	26990940	5.7
CPRIMES	O	636	600	95	3357822	25041190	7.5
ICHI	P	2307	2114	71	4438696	25454780	5.7
ICPRIMES	Q	552	512	100	3115997	43474570	14.0
SUMSQRT	R	707	604	138	4303205	41643530	9.7

*See Appendix F for test set contents.

Examining the last column of Table XIX, and comparing these values to those of Table XVII, it can be seen that the aggregate performance of TUMS is much closer to its best-case performance than to its worst-case. This good performance is not surprising since the worst-case performance wherein *every* statement of a program executes in its *slow* form cannot occur in practice since a mutant has only a single mutated statement.

Of greatest significance is the observation that the aggregate execution performance of schema-based TUMS is much better than that of the CX interpreter. Although not absolutely conclusive, these empirical results are strongly suggestive—they suggest that mutation analysis using the new MSG method is much faster than using the conventional interpretive method.

CHAPTER V

CONCLUSIONS AND FUTURE DIRECTIONS

Test data adequacy criteria serve as rules to determine whether a test set adequately tests a program. The effectiveness of a test data adequacy criterion is gauged by its ability to detect faults. Despite the empirically demonstrated efficacy of the mutation adequacy criterion, it has not seen much use because of the high operational cost conventionally incurred in evaluating the criterion. Our new method significantly reduces this operational cost. The Mutant Schema Generation (`MSG`) method we devised and the studies we have performed support our thesis that high performance mutation analysis is possible through the creation and instantiation of mutant schemata.

Chapter I of this dissertation established the context and importance of our research area. In Chapter II we presented the two components of the `MSG` method: the design of the mutant schemata and an approach for automatically generating metamutants. The `TUMS` prototype mutation analysis system, which we designed and implemented to allow us to empirically study the performance of `MSG`-based mutation analysis systems and gain insight into metamutant design concerns, was presented in Chapter III. The performance of the `TUMS` system was investigated in Chapter IV; included in Chapter IV are benchmark results that showed `TUMS` significantly faster than `Mothra`, a conventional interpretive mutation analysis system.

In the next section we discuss the advantages and disadvantages of using our method in constructing mutation analysis systems. We then offer an initial assessment of the significance of this work. We end this chapter with some suggestions for future work that stem from this dissertation research.

Advantages and Disadvantages

Besides the MSG approach described in this dissertation, the three other major approaches upon which to base a mutation analysis system are:

1. the *Interpretive* (or *Conventional*) *approach*,
2. the *Compiler-Integrated* (or *Machine Code Patching*) *approach*, and
3. the *Separate Compilation approach*.

The majority of mutation analysis systems built have been based on the interpretive approach [30, 31, 32, 41, 43, 52]. The most comprehensive mutation analysis system built to date, Mothra, is an interpreter-based system [33, 34]. Thus building a mutation analysis system using interpretive execution is the *conventional* approach. In what follows, we compare and contrast our approach to the conventional approach, but occasionally refer to the compiler-integrated and separate compilation approaches.

Advantages

Speed. Although a large number of metaprocedure function calls must be processed, MSG mutants run as compiled programs and thus execute at machine language speeds. Moreover, the use of twinning reduces the number of metaprocedure invocations. Consequently, as discussed in Chapter IV, MSG-based mutation analysis systems are faster than conventional systems.

Mutants also execute at machine language speeds in mutation analysis systems based on the compiler-integrated and separate compilation approaches. However, because the compilation of all mutant programs is folded into one single compilation, both MSG-based and compiler-integrated mutation analysis systems will outperform a system based on separate compilation unless mutant execution time greatly exceeds individual compilation (plus link) times. Krauser documented the speed advantage that comes from avoiding the compilation bottleneck in his comparison of the compiler-integrated CMothra system versus the separate compilation-based PMothra system [47].

Partial implementations possible. Since the ability to compile and run a program P is provided by an existing standard compiler, it is possible for an `MSG`-based 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. 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 allows quicker application of some aspects of mutation testing. This characteristic also gives greater freedom to experiment with the mutagenic operators.

Reduced implementation effort. The `MSG` method leaves the problems of providing run-time semantics and environment to separate tools (i.e., the compiler and run-time libraries of the given language and operating system). However, interpretive and compiler-integrated systems must deal with these complex and sophisticated implementation issues. Compiler-integrated systems, in particular, are demanding to build.[23]

Same operational environment. The `MSG` method permits testing to take place using the same compiler and environment that will be used by the program under test. Hence the program retains much of its original operational behavior. Only the separate compilation approach shares this advantage.

Portable. Because `MSG`-based mutation analysis systems operate at the source-level, they are easily ported. For example, a network computer system with different architectures (for example, Sun3s and Sun4s) could be utilized simply by recompiling the `MSG`-based system for each different type of machine.

Easy instrumentation. In `MSG`, the design of metamutants lends itself readily to adding instrumentation to the mutants (via expanded metaprocedures) to permit additional

[23]The `CMothra` system, the only extant compiler-integrated system, is approximately 115,000 lines of C source code. A prototype system, it supports only the *statement deletion* (`SSDL`) and the *scalar reference replacement* (`Vsrr`) mutagens.

monitoring of a mutant's execution state. This additional information could be used, for example, to implement weak mutation analysis. *Weak mutation* analysis [66, 67] employs a test data criterion less stringent than standard, or *strong*, mutation analysis: a mutant is killed as soon as the mutated component (e.g., `2 * A`) produces a result different from the original component (e.g., `2 + A`). Such state monitoring is easy in interpreter-based systems but would be difficult to achieve in systems based on the compiler-integrated or separate compilation approaches.

Disadvantages

Run-time overhead. The computational cost of performing execution accounting and invoking metaprocedures within a metamutant is not negligible. Machine code patched mutants do not have this overhead. Hence compiler-integrated systems are potentially faster than `MSG`-based systems but are slower than the hypothetical `IDEAL` presented in Chapter IV. Thus, although we do not know how much, if any, faster a compiler-integrated system might be versus an `MSG`-based system, we know that it will be bounded by the `IDEAL` values given in Table XIX.

Not applicable to all languages. There are a few languages, like BASIC, that lack the language features needed to express a mutant schema.[24] Consequently `MSG`-based mutation analysis systems cannot be built for these languages. The other approaches do not have this limitation.

Awkward to implement some mutagens. Metamutations representing certain mutagenic operators may be hard to represent compactly and efficiently. For example, it is very difficult to change evaluation order in expressions as a result of binary operator mutations that introduce operators at new precedence levels. Current interpretive systems, such as `Mothra`, suffer from the same problem, but compiler-integrated and separate compilation systems probably do not.

[24] In fact, BASIC is the only commonly used programming language we are aware of that lacks these language features.

Significance

The major consequence of this research is the introduction to the field of software testing of a fourth major approach to performing mutation analysis. This approach provides the basis for a new generation of mutation testing systems. The efficiency and flexibility of these new systems will allow a wider range of software to be tested than is now practical. In particular, software with high computation demands (i.e., software that takes an inherently long time to run) that could not previously be tested in a reasonable time period may now be amenable to testing. With the potential for an order-of-magnitude speed-up over existing systems, our approach is a significant step toward making mutation testing practical.

Another consequence of this research is an improved understanding of mutagens and program neighborhoods. In particular, this research has suggested changes to the set of valid C mutagens.

As a product of this research, a valuable research tool was created: the `TUMS` prototype mutation analysis system. Given this tool, it will now be possible to experiment with mutation testing in the C language environment.

Future Work

There are ample opportunities for extending this dissertation research; among the possibilities:

- *Hybrid approaches.* It is interesting to note that the `MSG` method is orthogonal to many of the approaches discussed in the section Related Work in Chapter I. 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. Also, there is no reason to believe that `MSG` systems could not be successfully adapted to run in a distributed computing environment. Exploring ways of combining these currently disparate approaches is likely to be very fruitful.

- *Incorporation of Weak Mutation.* As suggested earlier, it should be possible to incorporate weak mutation into the MSG approach.

- *Mutagen Effectiveness studies.* There is virtually no experience in C mutation-based testing. Using TUMS it should be possible to gain such experience and empirically answer questions like: what faults are not modeled by existing mutagens and which mutagens are best at insuring faults are uncovered?

  Research on selective mutation seeks to answer similar questions [39, 28]. It would be particularly exciting if an MSG-based mutation analysis system needed only to support the restricted set of mutagens suggested by selective mutation. Since this restricted set does not include any reference replacement mutagens, the need for dynamic typing would disappear. Since the overhead imposed by dynamic typing accounts for a large fraction of a metamutant's workload, a mutation analysis system based on selective mutation would exhibit improved execution efficiency.

- *Prototype Extension.* Although TUMS implements a very large subset of the C language and a considerable number of the G_2 mutagens, it is nonetheless a prototype system. Capitalizing on this existing software, a more comprehensive system, akin to the Mothra system, could be built. With greater capability and greater distribution, this system could advance the field of software testing in much the same way that the availability of the Mothra system did.

- *Mutant Spanning Sets.* Many of the mutant descriptors that are currently generated describe mutants that are semantically identical. Introducing some form of data flow analysis when creating and traversing the abstract syntax trees produced by mutant schema generation should yield information that could be used to partition the mutants into spanning sets, that is, into sets of mutants that behave identically for specific inputs. Using this information, only one representative of the spanning set would need to be executed to determine whether or not all the members of the spanning set were killed. This would reduce the cost of mutation analysis even further.

- *Metamutation Cost Model.* In our current `MSG` approach, *all* program statements are metamutated to create *one* metamutant capable of representing the *entire* program neighborhood—let us call this a *total neighborhood* metamutant. It is possible to select only *some* program statements to metamutate creating a metamutant that represents only *parts* of the program neighborhood, let us call this a *partial neighborhood* metamutant. In mutation analysis, the cost of compiling the total neighborhood metamutant is amortized over all the mutant executions. If we attempt to do mutation analysis with a collection of partial neighborhood metamutants, the compilation cost is spread over a smaller number of mutants. (In the extreme, this becomes the separate compilation approach.) On the other hand, partial neighborhood metamutants will execute more quickly than the total neighborhood metamutant.

  Research into developing a metamutation cost model that somehow relates the increasing cost of compilations to the decreasing cost of executions would provide the basis of a new technique for optimizing the execution of mutants and thus reducing the cost of mutation analysis. This same cost model could perhaps be extended to consider the cost of distributed mutant execution.

APPENDICES

Appendix A

Sample Metamutant

```
/* === MakeNeighborhood generated source begins: === */
#include "Mutant.h"

typedef struct {
     float N;
     float (*SUM);
     float NUMBER;
     float SQRT;
     float GUESS;
     float DELTA;
     float EPS;
} LOCAL_;

/* Run-time "Save Area" Variables */
tResult_ Result_[146];
tREF_    Left_[146];
/* "EXTERNed" Variables Used Also by Harness Routines */
long     Headway_ = 0;
long     HeadwayLimit_ = 0;
long     StmtTally_[18] = {0};
long     MaxStmt_ = 17;

tREF_
REF_(LOCAL_ * L_, tVariant_ Original, tPoint_ ChangePoint)
{
    tREF_     ref;
    tVariant_ Reference;

    static tDBL_  const10_ = 0.001;
    static tDBL_  const11_ = 0.0;
    static tDBL_  const12_ = 1.0;
    static tDBL_  const13_ = 2.0;

    if (ChangePoint==Mutant_.ChangePoint)
        Reference = Mutant_.Variation;
    else
        Reference = Original;

#define SETREF(ID,TYPE,INDR)  ref.addr = (tPTR_) &ID; \
  ref.type = TYPE;  ref.indr = INDR;
#define SETARR(ID,TYPE,INDR)  ref.addr = (tPTR_) &Result_[ChangePoint]; \
  Result_[ChangePoint].PTR_ = (tPTR_) &ID; \
  ref.type = TYPE;  ref.indr = INDR;

    switch (Reference)
    {
    case   3:  SETREF( L_->N,       FLT_,   0 );    break;
    case   4:  SETREF( L_->SUM,     FLT_,   1 );    break;
    case   5:  SETREF( L_->NUMBER,  FLT_,   0 );    break;
    case   6:  SETREF( L_->SQRT,    FLT_,   0 );    break;
    case   7:  SETREF( L_->GUESS,   FLT_,   0 );    break;
    case   8:  SETREF( L_->DELTA,   FLT_,   0 );    break;
    case   9:  SETREF( L_->EPS,     FLT_,   0 );    break;
    case  10:  SETREF( const10_,    DBL_,   0 );    break;
    case  11:  SETREF( const11_,    DBL_,   0 );    break;
    case  12:  SETREF( const12_,    DBL_,   0 );    break;
    case  13:  SETREF( const13_,    DBL_,   0 );    break;
    case  14:  SETREF( *L_->SUM,    FLT_,   0 );    break;
    case  15:  ref = UO_(REF_(L_,4,19),28,18);      break;
    case  16:  ref = UO_(REF_(L_,4,129),28,128);    break;
    case  17:  ref = UO_(REF_(L_,4,133),28,132);    break;
    default:   ERROR_("Illegal Reference Variant");
               STOP_();                             break;
    }

#undef SETREF

    return ref;
}
```

```
#define GOTO_(org,cp) switch \
  (cp==Mutant_.ChangePoint?Mutant_.Variation:org) { \
}

#define CONTINUE_(org,cp) {if (cp!=Mutant_.ChangePoint) \
    continue; \
else        \
    break;}

#define BREAK_(org,cp) {if (cp!=Mutant_.ChangePoint) \
    break;    \
else        \
    continue;}

/* === Metamutant Begin ========================== */
void SUMSQRT(
 float N_PARM_,
 float (*SUM_PARM_)
 )
{
 LOCAL_  ENV_;
 LOCAL_  *L_ = &ENV_;
 /* "Zero" the environment */
 (void) memset(&ENV_,   0, sizeof ENV_);
 /* "Zero" the "Save Areas" */
 (void) memset(Result_, 0, sizeof Result_);
 (void) memset(Left_,   0, sizeof Left_);
 /* Formal parameters => local equivalents */
 L_->N = N_PARM_;
 L_->SUM = SUM_PARM_;
if (6!=Mutant_.ChangePoint)
{ /*BEGIN 1*/ StmtTally_[1]++;  Headway_++;
if (1==Mutant_.StmtID)
    BO_(REF_(L_,9,10),REF_(L_,10,11),37,9);
else
    L_->EPS = 0.001;
}   /*END 1*/
if (13!=Mutant_.ChangePoint)
{ /*BEGIN 2*/ StmtTally_[2]++;  Headway_++;
if (2==Mutant_.StmtID)
    BO_(REF_(L_,15,17),REF_(L_,11,20),37,16);
else
    *L_->SUM = 0.0;
}   /*END 2*/
if (22!=Mutant_.ChangePoint)
{ /*BEGIN 3*/ StmtTally_[3]++;  Headway_++;
if (3==Mutant_.StmtID)
    BO_(REF_(L_,5,26),REF_(L_,12,27),37,25);
else
    L_->NUMBER = 1.0;
}   /*END 3*/
if (29!=Mutant_.ChangePoint)
{ /*BEGIN 4*/ StmtTally_[4]++;  Headway_++;
while(LOOP_(0,31)||(4==Mutant_.StmtID?PRED_(BO_(REF_(L_,5,34),REF_(L_,
3,35),10,33),0,32):L_->NUMBER <= L_->N))
{
if (Headway_ > HeadwayLimit_) HeadwayExceeded_();
StmtTally_[4]++;  Headway_++;
    if (36!=Mutant_.ChangePoint)
    { /*BEGIN 5*/ StmtTally_[5]++;  Headway_++;
    {
    if (40!=Mutant_.ChangePoint)
    { /*BEGIN 6*/ StmtTally_[6]++;  Headway_++;
    if (6==Mutant_.StmtID)
        BO_(REF_(L_,7,44),BO_(BO_(REF_(L_,5,47),REF_(L_,13,48),17,46),
        REF_(L_,12,49),14,45),37,43);
    else
        L_->GUESS = L_->NUMBER / 2.0 + 1.0;
    }   /*END 6*/
```

```
if (51!=Mutant_.ChangePoint)
{ /*BEGIN 7*/ StmtTally_[7]++;  Headway_++;
if (7==Mutant_.StmtID)
    BO_(REF_(L_,6,55),REF_(L_,11,56),37,54);
else
    L_->SQRT = 0.0;
}   /*END 7*/
if (58!=Mutant_.ChangePoint)
{ /*BEGIN 8*/ StmtTally_[8]++;  Headway_++;
if (8==Mutant_.StmtID)
    BO_(REF_(L_,8,62),BO_(REF_(L_,7,64),REF_(L_,6,65),15,63),37,
    61);
else
    L_->DELTA = L_->GUESS - L_->SQRT;
}   /*END 8*/
if (67!=Mutant_.ChangePoint)
{ /*BEGIN 9*/ StmtTally_[9]++;  Headway_++;
while(LOOP_(0,69)||(9==Mutant_.StmtID?PRED_(BO_(REF_(L_,8,72),REF_(L_,
9,73),9,71),0,70):L_->DELTA >  L_->EPS))
{
if (Headway_ > HeadwayLimit_) HeadwayExceeded_();
StmtTally_[9]++;  Headway_++;
    if (74!=Mutant_.ChangePoint)
    { /*BEGIN 10*/ StmtTally_[10]++;  Headway_++;
    {
    if (78!=Mutant_.ChangePoint)
    { /*BEGIN 11*/ StmtTally_[11]++;  Headway_++;
    if (11==Mutant_.StmtID)
        BO_(REF_(L_,6,82),REF_(L_,7,83),37,81);
    else
        L_->SQRT = L_->GUESS;
    }   /*END 11*/
    if (85!=Mutant_.ChangePoint)
    { /*BEGIN 12*/ StmtTally_[12]++;  Headway_++;
    if (12==Mutant_.StmtID)
        BO_(REF_(L_,7,89),BO_(BO_(REF_(L_,6,92),BO_(REF_(L_,5,94),
        REF_(L_,6,95),17,93),14,91),REF_(L_,13,96),17,90),37,88);
    else
        L_->GUESS = (L_->SQRT + L_->NUMBER / L_->SQRT) / 2.0;
    }   /*END 12*/
    if (98!=Mutant_.ChangePoint)
    { /*BEGIN 13*/ StmtTally_[13]++;  Headway_++;
    if (13==Mutant_.StmtID)
        BO_(REF_(L_,8,102),BO_(REF_(L_,7,104),REF_(L_,6,105),15,
        103),37,101);
    else
        L_->DELTA = L_->GUESS - L_->SQRT;
    }   /*END 13*/
    if (107!=Mutant_.ChangePoint)
    { /*BEGIN 14*/ StmtTally_[14]++;  Headway_++;
    if((14==Mutant_.StmtID?PRED_(BO_(REF_(L_,8,112),REF_(L_,11,
    113),8,111),0,110):L_->DELTA < 0.0))
    {
        if (114!=Mutant_.ChangePoint)
        { /*BEGIN 15*/ StmtTally_[15]++;  Headway_++;
        if (15==Mutant_.StmtID)
            BO_(REF_(L_,8,118),UO_(REF_(L_,8,120),30,119),37,117);
        else
            L_->DELTA = -L_->DELTA;
        }   /*END 15*/
    }
    }   /*END 14*/
    }
    }   /*END 10*/
/*END LOOP*/ }
}   /*END 9*/
```

```
    if (123!=Mutant_.ChangePoint)
    { /*BEGIN 16*/ StmtTally_[16]++;  Headway_++;
    if (16==Mutant_.StmtID)
        BO_(REF_(L_,16,127),BO_(REF_(L_,17,131),REF_(L_,6,134),14,130),
        37,126);
    else
        *L_->SUM = *L_->SUM + L_->SQRT;
    }   /*END 16*/
    if (136!=Mutant_.ChangePoint)
    { /*BEGIN 17*/ StmtTally_[17]++;  Headway_++;
    if (17==Mutant_.StmtID)
        BO_(REF_(L_,5,140),BO_(REF_(L_,5,142),REF_(L_,12,143),14,141),
        37,139);
    else
        L_->NUMBER = L_->NUMBER + 1.0;
    }   /*END 17*/
    }
    }   /*END 5*/
/*END LOOP*/ }
}   /*END 4*/
}
/* === Metamutant End =========================== */
/* === MakeNeighborhood generated source ends. === */
```

Appendix B

Example Interface Files

Interface files for SUMSQRT.c

```
/* ====> VarDecl.h: <==== */
/* Prototype of function being mutated */
void SUMSQRT( float, float * );

/* (Return value and) arguments of the function being mutated */
typedef struct {
        float   N;
        float   SUM;
} tARG_;
```

```
/* ====> Call.c <==== */
SUMSQRT(ARG_.N, &(ARG_.SUM));
```

```
/* ====> Compare.c <==== */

#define epsilon(a,b,eps)  ( (a)>=(b) ? ((a)-(b))<(eps) : ((b)-(a))<(eps) )
AND  epsilon( ARG_.SUM, POST_.SUM, 0.001)
#undef epsilon
```

```
/* ====> ReadTC.c  <==== */
"%f%f%f%f%d%ld%ld%d"
,&TC_.Ante.N
,&TC_.Ante.SUM
,&TC_.Post.N
,&TC_.Post.SUM
/* --- do not modify following: --- */
,&TC_.Status_
,&TC_.Headway_
,&TC_.CPUtime_
,&TC_.tcNum_
);
```

```
/* ====> WriteTC.c  <==== */
"%f %f %f %f %d %ld %ld %d \n"
,TC_.Ante.N
,TC_.Ante.SUM
,TC_.Post.N
,TC_.Post.SUM
/* --- do not modify following: --- */
,TC_.Status_
,TC_.Headway_
,TC_.CPUtime_
,TC_.tcNum_
);
```

Appendix C

Analyze Driver Programs

```
                   /* Analyze.h - Analyze header file */

#ifndef Analyze_h_           /* To prevent problems from multiple inclusions */
#define Analyze_h_

#include <sys/time.h>

#include "tums.h"
#include "Mutant.h"

/* --- Macros (to unify the interface) ------------------------------------ */
#define  ARG_  Rbuff->Arg_
#define  POST_ Pbuff->TC_.Post

/* --- VarDecl.h -(from $Neighborhood+$TestSet directory)------------------ */
#include "VarDecl.h"

/* --- Type Declarations for the Buffer Structures ------------------------ */

typedef    /*  Test Case structure */
    struct {
        tARG_  Ante;
        tARG_  Post;
        int    Status_;
        long   Headway_;
        long   CPUtime_;
        int    tcNum_;
    }
tTC_;

typedef    /* Pair Buffer structure */
    struct {
        boolean             Continue_;
        tTC_                TC_;
        long                HeadwayLimit_;
        struct itimerval    TimeLimit_;
        tMutantDescriptor_  Mutant_;
        boolean             anVerbose_;
    }
tPB_;

typedef    /* Results Buffer structure */
    struct {
        tMutStatus_  Result_;
        tARG_        Arg_;
    }
tRB_;

#endif
```

```
                /* Analyze.c - Analyze Neighborhood main */

#define MAIN

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <signal.h>
/* #include <siginfo.h>   // needed for Solaris because of psignal() */
#include <string.h>

#include <errno.h>

#include <unistd.h>
#include <sys/wait.h>

#include <sys/time.h>
#include <sys/resource.h>
int   setitimer(int which, struct itimerval *value, struct itimerval *ovalue);
int   getrusage(int who, struct rusage *rusage);
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "tums.h"
#include "Mutant.h"
#include "IPC.h"
#include "Art.h"

#include "Analyze.h"


/* --- Constants ---------------------------------------------------------- */
#define TimeOutMultiplier_ 10
#define SlowTwinSlowDown_ 40
#define AP_NAME "AnalyzePair"


/* --- Global Variables --------------------------------------------------- */

/* Semaphore IDs and constants:                                             */
enum {NP,    /* NP = iN Pair buffer semaphore                               */
      EP,    /* EP = Empty Pair buffer semaphore                            */
      NR};   /* NR = iN Results buffer semaphore                            */
int     sem[3] = {INT_MIN,INT_MIN,INT_MIN};       /* Arbitrary "ID" init   */
char semstr[3][10];                               /* sem values as strings */

/* Shared Memory IDs and constants:                                         */
enum {PB,    /* PB = Pair Buffer (shared memory)                            */
      RB};   /* RB = Results Buffer (shared memory)                         */
int     shm[2] = {INT_MIN,INT_MIN};               /* Arbitrary "ID" init   */
char shmstr[2][10];                               /* shm values as strings */

/* Pointers to the shared memory buffers                                    */
tPB_  *  Pbuff = NULL;                            /* Pair buffer pointer   */
tRB_  *  Rbuff = NULL;                            /* Results buffer pointer*/


/* Child (AnalyzePair) process pid                                          */
int      pid;
```

```
/* --- Functions ---------------------------------------------------------- */

void
SkipHdr( FILE * input )
{
    int ch, l;

    for (l=1;l<=2;l++)
        while ( (ch=getc(input)) != EOF )
            if (ch=='\n')
                break;
}


void
CleanupIPC(int sig)
{
    sem[NP]!=-1     ? semctl(sem[NP], 0,  IPC_RMID, 0) : 0;
    sem[EP]!=-1     ? semctl(sem[EP], 0,  IPC_RMID, 0) : 0;
    sem[NR]!=-1     ? semctl(sem[NR], 0,  IPC_RMID, 0) : 0;
    shm[PB]!=-1     ? shmctl(shm[PB],     IPC_RMID, 0) : 0;
    shm[RB]!=-1     ? shmctl(shm[RB],     IPC_RMID, 0) : 0;
    if (sig==0)
        return;
    else
    {
        (void) fprintf(stderr,
          "Error: Analyze received signal %d", sig);
        (void) psignal(sig, " ");

        /* Explicitly signal child (AnalyzePair) to die. */
        kill(pid, SIGKILL);

        exit(FAILURE);
    }
}


int
main( int argc, char **argv )
{
    FILE *  ti_;                               /* Test Set input file handle */
    char    TSname[MAX_FILE_NAME_LENGTH+1];    /* Test Set filename          */
    char    Dname[MAX_FILE_NAME_LENGTH+1];     /* Mutant Descriptor filename */
    char    ARTname[MAX_FILE_NAME_LENGTH+1];   /* Analysis Results filename  */
    int     Narg, Targ;                        /* argv[] subscripts          */
    boolean anVerbose_ = FALSE;                /* Verbosity flag             */

    int          sig;
    int          return_status, return_code;
    char *       ap_args[7];                   /* AnalyzePair argument vector*/

    int          tc, final_tc;
    tTC_         TC_;
    tMutantID_   mutant, NumberOfMutants;
    tMutStatus_  Result_;
    tART         ART;                          /* Analysis Results Table     */
```

```
/* Timer related: */
struct rusage     anTime0_;           /* Analyze program start time    */
struct rusage     anTime1_;           /* Analyze program finish time   */
long              anElapsedTime_;     /* Analyze in milliseconds       */
struct rusage     apTime0_;           /* AnalyzePair     start time    */
struct rusage     apTime1_;           /* AnalyzePair     finish time   */
long              apElapsedTime_;     /* Analyze in milliseconds       */
long              TotalElapsedTime_;  /* Analyze+AnalyzePair in ms.    */

long              HeadwayLimit_;      /* Mutant headway limit in stmts */
int               TimeOut;            /* Mutant time limit in ms       */
struct itimerval  TimeLimit_;         /* Mutant time limit structure   */


/* --- Start analysis execution --- */
getrusage(RUSAGE_SELF,     &anTime0_);
getrusage(RUSAGE_CHILDREN, &apTime0_);
(void) fprintf(stderr, "\nStarting...\n\n");

/* Process command-line arguments. */
if (argc < 3)
{
    ERROR("Missing command-line argument(s) to Analyze");
    exit(FAILURE);
}
/* Check for any options that may have been specified. */
if (argv[1][0]=='-')
{
    Narg = 2;
    Targ = 3;
    if (strchr(argv[1],'v')) anVerbose_ = TRUE;
}
else
{
    Narg = 1;
    Targ = 2;
}

/* Open Test Set file */
strcpy(TSname,  argv[Narg]);
strcat(TSname, "+");
strcat(TSname,  argv[Targ]);
strcat(TSname, "/");
strcat(TSname, argv[Targ]);
strcat(TSname, ".ts");
ti_ = fopen(TSname, "r");
if ( ti_ == NULL )
{
    OS_ERROR("fopen of test set file");
    exit(errno);
}
SkipHdr( ti_ );

/* Obtain Mutant Descriptors (from file) */
strcpy(Dname,  argv[Narg]);
strcat(Dname,  ".md");
NumberOfMutants = GetMutants_( Dname );
if (anVerbose_)
    (void) fprintf(stderr, "%d mutant descriptors found \n\n",
      NumberOfMutants);
```

```
/* Ready the Analysis Results Table (ART) */
strcpy(ARTname,  argv[Narg]);
strcat(ARTname, "+");
strcat(ARTname,  argv[Targ]);
strcat(ARTname, "/");
strcat(ARTname,  argv[Targ]);
strcat(ARTname, ".art");
if ( !OpenART( &ART, ARTname ) )
{
    ERROR("Unable to open ART");
    exit(FAILURE);
}

/* Cause all signals to be caught by the "CleanupIPC" routine.
 * The specific number of signals is somewhat system dependent;
 * the top value (NSIG from signal.h) may need adjustment.
 * NOTE: this implementation assumes that "Reliable Signal"
 * semantics are implemented with signal().  If this isn't true,
 * the sigaction() routine will need to be used.  See Stevens,
 * Chapter 10 (especially pp. 296-299) for more information.
*/
for (sig=1; sig < NSIG; sig++)
{
    switch(sig)
    {
    /* Don't try to catch these signals    */
    case SIGKILL: case SIGSTOP: case SIGCHLD:
        break;
    default:
        if ( signal(sig, CleanupIPC)==SIG_ERR)
        {
            (void) fprintf(stderr,
              "Error: unable to catch signal %d", sig);
            (void) psignal(sig, " ");
        }
        break;
    }
}

/* Set-up semaphores. */
sem[NP] = SemaphoreInit(0);  /* Number of Pairs in Pair buffer.      */
sem[EP] = SemaphoreInit(1);  /* "Emptiness" of Pair buffer.          */
sem[NR] = SemaphoreInit(0);  /* Number of Results in Results buffer. */

/* Set-up shared memory. */
shm[PB] = shmget(IPC_PRIVATE, sizeof(tPB_), 0600);
shm[RB] = shmget(IPC_PRIVATE, sizeof(tRB_), 0600);
Pbuff   = (tPB_ *)  shmat(shm[PB], NULL, 0         );
Rbuff   = (tRB_ *)  shmat(shm[RB], NULL, SHM_RDONLY);
```

```
/*
 * Set up argument list for invocation of AnalyzePair.
 * Note that the following assignments merely assign to the
 * ap_args array the addresses where the argument values
 * will be found and NOT the actual argument values themselves.
*/
ap_args[0] = AP_NAME;       /* Name of child executable image. */
ap_args[1] = semstr[NP];
ap_args[2] = semstr[EP];
ap_args[3] = semstr[NR];
ap_args[4] = shmstr[PB];
ap_args[5] = shmstr[RB];
ap_args[6] = NULL;

sprintf(semstr[NP], "%d", sem[NP]);
sprintf(semstr[EP], "%d", sem[EP]);
sprintf(semstr[NR], "%d", sem[NR]);
sprintf(shmstr[PB], "%d", shm[PB]);
sprintf(shmstr[RB], "%d", shm[RB]);


/* --- Invoke the AnalyzePair routine --- */
if ( (pid = fork()) < 0)
{
    OS_ERROR("Can't fork to run AnalyzePair.");
    exit(errno);
}
else
{
    if (pid == 0)
    {   /* Child process:  run AnalyzePair. */
        execvp(AP_NAME,ap_args);
        OS_ERROR("Invocation of AnalyzePair failed.");
        _exit( FAILURE );
    }
}

/* From this point on, we are running concurrently with AnalyzePair. */


/* --- Main processing loop.  Process BY Test Case, BY Mutant. --- */
Result_ = UNKNOWN_;
final_tc = MaxTestCase(ART);
for (tc=1; tc <= final_tc  AND  Result_ != ABORT_; tc++)
{
    /* Obtain next Test Case TC_. */
    if (anVerbose_)
        (void) fprintf(stderr, "--Analyzing Test Case %d\n", tc);
    (void) fscanf(ti_,
    #include "ReadTC.c"
    while ( (TC_.tcNum_ < tc) AND !feof(ti_) )
    {
        (void) fscanf(ti_,
        #include "ReadTC.c"
    }
    if (TC_.tcNum_ != tc)
    {
        ERROR("Unable to obtain requested test case");
        exit(FAILURE);
    }
```

```
    /* Set workload limit based on original program's headway on this tc.*/
    HeadwayLimit_ = TC_.Headway_ * TimeOutMultiplier_;

    /* As insurance, set a virtual timer.  A mutant program that is running
     * "too long" should really halt when its internal Headway_ counter
     * exceeds the calculated HeadwayLimit_. This virtual timer is being
     * set just in case that doesn't happen.  (Besides, the code was
     * already in place.)
     * It is an ERROR in the metamutant's headway bookkeeping if this timer
     * causes execution to halt; the reason should be investigated.
     * The SlowTwinSlowDown_ multiplier attempts to adjust the
     * CPUtime by the amount of slowdown caused by Slow Twin execution.
     */
    TimeOut = (TC_.CPUtime_ * SlowTwinSlowDown_) * TimeOutMultiplier_;
    TimeLimit_.it_interval.tv_sec  = 0;
    TimeLimit_.it_interval.tv_usec = 0;
    TimeLimit_.it_value.tv_sec     =  TimeOut / 1000;
    TimeLimit_.it_value.tv_usec    = (TimeOut % 1000) * 1000;


    /* Process By Mutant. */
    for (mutant=1; mutant <= NumberOfMutants; mutant++)
    {
        if (MutantStatus(ART, mutant) == LIVE_)
            if (Status(ART, tc, mutant) == UNKNOWN_)
            {
                /* Enter information into Pair buffer for AnalyzePair. */
                Pwait(sem[EP]);
                    Pbuff->Continue_      = TRUE;
                    Pbuff->TC_            = TC_;
                    Pbuff->HeadwayLimit_ = HeadwayLimit_;
                    Pbuff->TimeLimit_     = TimeLimit_;
                    Pbuff->Mutant_        = ReferenceMutant_(mutant);
                    Pbuff->anVerbose_     = anVerbose_;
                Vsignal(sem[NP]);

                /* === Give AnalyzePair a chance to work === */

                Pwait(sem[NR]);    /* Wait until AnalyzePair has results */
                Result_ = Rbuff->Result_;


                /* Use the results to update the ART. */
                if      (Result_ == ABORT_)
                    break;
                else if (Result_ == DEAD_)
                    Kill(ART, tc, mutant);
                else
                    MarkPair(ART, tc, mutant, Result_);
            }
    }

}
```

```
    /* Finalizations */

    /* Tell AnalyzePair child to quit. */
    Pwait(sem[EP]);
        Pbuff->Continue_     = FALSE;
        Pbuff->anVerbose_    = anVerbose_;
    Vsignal(sem[NP]);

    /* Wait until AnalyzePair finished. */
    waitpid(pid, &return_status, 0);
    /* return_code = WIFEXITED(return_status);  // don't currently need this */

    CleanupIPC( 0 );

    if ( !CloseART(ART, ARTname) )
        ERROR("ART update failed");

    if ( fclose(ti_) )
        OS_ERROR("fclose of test case file");

    /* Determine elapsed times. */
    getrusage(RUSAGE_SELF,     &anTime1_);
    anElapsedTime_ =
      (1000.0*anTime1_.ru_utime.tv_sec + .001*anTime1_.ru_utime.tv_usec
    +  1000.0*anTime1_.ru_stime.tv_sec + .001*anTime1_.ru_stime.tv_usec)
    -
      (1000.0*anTime0_.ru_utime.tv_sec + .001*anTime0_.ru_utime.tv_usec
    +  1000.0*anTime0_.ru_stime.tv_sec + .001*anTime0_.ru_stime.tv_usec);

    getrusage(RUSAGE_CHILDREN, &apTime1_);
    apElapsedTime_ =
      (1000.0*apTime1_.ru_utime.tv_sec + .001*apTime1_.ru_utime.tv_usec
    +  1000.0*apTime1_.ru_stime.tv_sec + .001*apTime1_.ru_stime.tv_usec)
    -
      (1000.0*apTime0_.ru_utime.tv_sec + .001*apTime0_.ru_utime.tv_usec
    +  1000.0*apTime0_.ru_stime.tv_sec + .001*apTime0_.ru_stime.tv_usec);

    TotalElapsedTime_ = anElapsedTime_ + apElapsedTime_;
    TotalElapsedTime_ = (TotalElapsedTime_ < 10) ? 10 : TotalElapsedTime_;

    (void) fprintf(stderr, "\n"
      "...Ending.         (Analyze elapsed time = %4ld milliseconds)\n"
      "               (AnalyzePair elapsed time = %4ld milliseconds)\n"
      "            (Total combined elapsed time = %4ld milliseconds)\n\n",
      anElapsedTime_, apElapsedTime_, TotalElapsedTime_ );

    exit(0);
}
```

```
     /* AnalyzePair.c - Analyze the given "test case"--"mutant" pair.  */

#define MAIN

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <signal.h>
/* #include <siginfo.h>   // needed for Solaris because of psignal() */
#include <string.h>
#include <setjmp.h>
#include <sys/time.h>
#include <sys/resource.h>
int   setitimer(int which, struct itimerval *value, struct itimerval *ovalue);
int   getrusage(int who, struct rusage *rusage);
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "tums.h"
#include "IPC.h"
#include "Mutant.h"

#include "Analyze.h"


/* --- External Variables -(from system)---------------------------------- */
extern char *sys_siglist[];

/* --- External Variables -(used to communicate with Metamutant)----------- */
extern long  Headway_;
extern long  HeadwayLimit_;
extern long  StmtTally_[];
extern long  MaxStmt_;


/* --- Global Variables --------------------------------------------------- */

/* Semaphore IDs and constants:                                            */
enum {NP,    /* NP = iN Pair buffer semaphore                              */
      EP,    /* EP = Empty Pair buffer semaphore                           */
      NR};   /* NR = iN Results buffer semaphore                           */
int     sem[3] = {INT_MIN,INT_MIN,INT_MIN};       /* Arbitrary "ID" init   */

/* Shared Memory IDs and constants:                                        */
enum {PB,    /* PB = Pair Buffer (shared memory)                           */
      RB};   /* RB = Results Buffer (shared memory)                        */
int     shm[2] = {INT_MIN,INT_MIN};               /* Arbitrary "ID" init   */

/* Pointers to the shared memory buffers                                   */
tPB_  *  Pbuff = NULL;                            /* Pair buffer pointer   */
tRB_  *  Rbuff = NULL;                            /* Results buffer pointer*/


/* Timer related: */
struct rusage     Time0_ = {0};           /* Metamutant start  time        */
struct rusage     Time1_ = {0};           /* Metamutant finish time        */
long              ElapsedTime_ = 0;       /*   Per Metamutant,  in ms      */

struct itimerval  old_value_ = {0};       /* (dummy arg needed for call)   */
```

```
/* Signal related: */
int               sig = NSIG;
static sigjmp_buf jmpbuf      = {INT_MIN}; /* for sigsetjmp/siglongjmp      */
static boolean    jmpbufVALID = FALSE;     /* enable/disable jump back      */

/* Misc. */
long              stmt = 0;                /* StmtTally_[] subscript        */
boolean           Different_ = TRUE;       /* Comparison to expected flag   */
char              SigMessage[80] = {'\0'}; /* Message from SigCatch routine */



/* --- Functions ---------------------------------------------------------- */

static void
SigCatch(int signum);



int
main( int argc, char * argv[] )
{
    /* Cause all signals to be caught by the "SigCatch" routine.
     * The specific number of signals is somewhat system dependent;
     * the top value (NSIG from signal.h) may need adjustment.
     * NOTE: this implementation assumes that "Reliable Signal"
     * semantics are implemented with signal().  If this isn't true,
     * the sigaction() routine will need to be used.  See Stevens,
     * Chapter 10 (especially pp. 296-299) for more information.
     */
    jmpbufVALID = FALSE;
    for (sig=1; sig < NSIG; sig++)
    {
        switch(sig)
        {
        /* Don't try to catch these signals    */
        case SIGKILL: case SIGSTOP:
            break;
        default:
            if ( signal(sig, SigCatch)==SIG_ERR)
            {
                (void) fprintf(stderr,
                  "Error: unable to catch signal %d", sig);
                (void) psignal(sig, " ");
            }
            break;
        }
    }

    /* Get Semaphore IDs (semid) and Shared Memory IDs (shmid). */
    if (argc < 6)
    {
        (void) fprintf(stderr,
          "Error: insufficient args to AnalyzePair\n");
        exit(ABORT_);
    }
    sem[NP] = atoi(argv[1]);
    sem[EP] = atoi(argv[2]);
    sem[NR] = atoi(argv[3]);
    shm[PB] = atoi(argv[4]);
    shm[RB] = atoi(argv[5]);
```

```
/* Connect shared memory segments to this address space. */
Pbuff  = (tPB_ *)  shmat(shm[PB], NULL, SHM_RDONLY);
Rbuff  = (tRB_ *)  shmat(shm[RB], NULL, 0         );


/* --- Invoke specified mutant using specified test case --- */

/* Each iteration of the loop processes a <TestCase,Mutant> pair. */
while ('1')
{
    Pwait(sem[NP]);    /* Wait here until Analyze fills Pair buffer */

    /* Retrieve next <TestCase,Mutant> pair from Pair buffer. */
    Rbuff->Arg_ = Pbuff->TC_.Ante;
    SetMutant_( &(Pbuff->Mutant_) );

    if ( !Pbuff->Continue_ )
        exit(SUCCESS);

    Different_ = TRUE;

    if ( sigsetjmp(jmpbuf,1) == 0 )
    {
        jmpbufVALID = TRUE;

        /* Initialize the workload and statement tallies. */
        HeadwayLimit_ = Pbuff->HeadwayLimit_;
        Headway_ = 0;
        for (stmt=1; stmt <= MaxStmt_; stmt++)
            StmtTally_[stmt] = 0;

        /* Set the virtual timer as "insurance". */
        setitimer(ITIMER_VIRTUAL, &(Pbuff->TimeLimit_), &old_value_);

        getrusage(RUSAGE_SELF, &Time0_);

        #include "Call.c"

        getrusage(RUSAGE_SELF, &Time1_);

        SigMessage[0] = '\0';

        /* Compare mutant results with expected */
        if ( TRUE
            #include "Compare.c"
        )
            Different_ = FALSE;
    }
    else /* return from longjmp as a result of a signal */
    {
        getrusage(RUSAGE_SELF, &Time1_);
        Different_ = TRUE;
    }


    ElapsedTime_ =
        (1000.0*Time1_.ru_utime.tv_sec + .001*Time1_.ru_utime.tv_usec
      +  1000.0*Time1_.ru_stime.tv_sec + .001*Time1_.ru_stime.tv_usec)
      -
        (1000.0*Time0_.ru_utime.tv_sec + .001*Time0_.ru_utime.tv_usec
      +  1000.0*Time0_.ru_stime.tv_sec + .001*Time0_.ru_stime.tv_usec);
    ElapsedTime_ = (ElapsedTime_ < 10)? 10 : ElapsedTime_;
```

```
        if (Different_)
        {
            /* Kill this mutant */
            if (Pbuff->anVerbose_)
            {
                (void) fprintf(stderr,
                  "  Mutant %4d     killed by tc %4d",
                  Pbuff->Mutant_.MutantID, Pbuff->TC_.tcNum_);
                (void) fprintf(stderr, "    (%s%4d %6ldms %8ld  <%d, %d>)\n",
                  Pbuff->Mutant_.MutagenCode, Pbuff->Mutant_.StmtID,
                  ElapsedTime_, Headway_,
                  Pbuff->Mutant_.Variation, Pbuff->Mutant_.ChangePoint);
                if ( !NULLS(SigMessage) )
                    (void) fprintf(stderr, "    %s\n", SigMessage);
            }
            Rbuff->Result_ = DEAD_;
        }
        else
        {
            if (Pbuff->anVerbose_)
            {

                (void) fprintf(stderr,
                  "  Mutant %4d NOT killed by tc %4d",
                  Pbuff->Mutant_.MutantID, Pbuff->TC_.tcNum_);
                (void) fprintf(stderr, "    (%s%4d %6ldms %8ld  <%d, %d>)\n",
                  Pbuff->Mutant_.MutagenCode, Pbuff->Mutant_.StmtID,
                  ElapsedTime_, Headway_,
                  Pbuff->Mutant_.Variation, Pbuff->Mutant_.ChangePoint);
            }
            Rbuff->Result_ = LIVE_;
        }

        Vsignal(sem[EP]);          /* Indicate done with Pair buffer */
        Vsignal(sem[NR]);          /* Indicate Result Buffer filled  */
    }  /* end while */
}
```

```
static void
SigCatch(int signum)
{
    if (!jmpbufVALID)
        /* Premature signal, ignore */
        return;

    /* Report on why this mutant will be marked as killed. */
    if (Pbuff->anVerbose_)
    {
        switch(signum)
        {
        case SIGPROF:
            (void) strcpy(SigMessage,
              "Time-out...Headway count exceeded");
        break;
        case SIGVTALRM:
            (void) strcpy(SigMessage,
              "Time-out...virtual alarm triggered  *TUMS ERROR*!");
        break;
        case SIGUSR1:
            (void) strcpy(SigMessage,
              "SIGUSR1 signal...indicating a controlled termination");
        break;
        case SIGUSR2:
            (void) strcpy(SigMessage,
              "SIGUSR2 signal...probably a bad switch value");
        break;
        case SIGILL:
            (void) strcpy(SigMessage,
              "SIGILL signal...illegal instruction");
        break;
        case SIGBUS:
            (void) strcpy(SigMessage,
              "SIGBUS signal...probable attempt to corrupt code");
        break;
        case SIGSEGV:
            (void) strcpy(SigMessage,
              "SIGSEGV signal...invalid address reference");
        break;
        case SIGXCPU:
            (void) strcpy(SigMessage,
              "SIGXCPU signal...exceeded CPU time limit");
        break;
        case SIGXFSZ:
            (void) strcpy(SigMessage,
              "SIGXFSZ signal...exceeded file size limit");
        break;
        case SIGFPE:
            (void) strcpy(SigMessage,
              "SIGFPE signal...floating point exception");
        break;
        case SIGINT:
            (void) strcpy(SigMessage,
              "SIGINT signal...aborting analysis");
            /* ??? need to set a flag or something */
        break;
        default:
            if (signum<=NSIG)
                (void) strcpy(SigMessage, sys_siglist[signum]);
            else
                (void) sprintf(SigMessage, "Caught unknown SIGNAL %d",
                  signum);
        }
    }

    /* Jump back to processing loop to conclude bookkeeping. */
    jmpbufVALID = FALSE;
    siglongjmp(jmpbuf, 1);
}
```

Appendix D

Experiment Specimen Programs

CHI—C implementation (chi.c)

```
/*
 * The random number generator and the code for calculating
 * the chi-square test come from Robert Sedgewick's "Algorithms (2nd ed)",
 * pp. 511-519, Addison-Wesley.
 */



#define R 100
                /* frequency tallies for chi-square calculation */
                /*      V                                      */
float CHI(int N, float *f)
{
    int t, p, p1, p0, val1, val2, val3, mod1, mod2, i;
    float ftt;

    /* Initialize tallies to zero */
    for (i=0; i<=R-1; i++)
        f[i] = 0;

    p = 1234567;
    for (i=1; i<=N; i++)
    {
        /* Generate a random number, t, using the Linear Congruential
         * Method.  The random number p is calculated each time through
         * the following code.  It is mapped to the range 0 through R-1
         * to produce the random number t.                           */
        p1 = p / 10000;
        p0 = p - (p1 * 10000);

        val1 = (p0 * 3141) + (p1 * 5821);
        mod1 = val1 - ((val1 / 10000) * 10000);

        val2 = (mod1 * 10000) + (p0 * 5821);
        mod2 = val2 - ((val2 / 100000000) * 100000000);

        val3 = mod2 + 1;
        p =    val3 - ((val3 / 100000000) * 100000000);

        /* Adjust to range 0 through R-1. */
        t = ((p / 10000)*R) / 10000;


        /* Tally number of times each random number is produced. */
        f[t] = f[t] + 1.0;
    }

    /* Perform a Chi-square test on the distribution of random
     * numbers produced by the scheme above.  If the chisquare
     * value is close to "R", then the numbers are "random".     */
    ftt = 0.0;
    for (i=0; i<=R-1; i++)
        ftt = ftt + (f[i]*f[i]);
    ftt = ((R*ftt)/N) - N;

    return ftt;
}
```

CHI—Fortran implementation (`chi.f`)

```
C*
C* The random number generator and the code for calculating
C* the chi-square test come from Robert Sedgewick's "Algorithms (2nd ed)",
C* pp. 511-519, Addison-Wesley.
C*

      REAL FUNCTION CHI(N)
      INTEGER N
C
      PARAMETER (R=100)
C
      INTEGER t, p, p1, p0, val1, val2, val3, mod1, mod2, I
C     REAL f(0:R)
      REAL f(0:100)
C          * frequency tallies for chi-square calculation
      REAL ftt

C     * Initialize tallies to zero
      DO 10 i=0,R-1
10        f(i) = 0

      p = 1234567
      DO 20 i=1,N
C
C         * Generate a random number, t, using the Linear Congruential
C         * Method.  The random number p is calculated each time through
C         * the following code.  It is mapped to the range 0 through R-1
C         * to produce the random number t.
          p1 = p / 10000
          p0 = p - (p1 * 10000)

          val1 = (p0 * 3141) + (p1 * 5821)
          mod1 = val1 - ((val1 / 10000) * 10000)

          val2 = (mod1 * 10000) + (p0 * 5821)
          mod2 = val2 - ((val2 / 100000000) * 100000000)

          val3 = mod2 + 1
          p =    val3 - ((val3 / 100000000) * 100000000)

C         * Adjust to range 0 through R-1. */
          t = ((p / 10000)*R) / 10000


C         * Tally number of times each random number is produced.
          f(t) = f(t) + 1.0
20        CONTINUE

C     * Perform a Chi-square test on the distribution of random
C     * numbers produced by the scheme above.  If the chisquare
C     * value is close to "R", then the numbers are "random".
      ftt = 0.0
      DO 30 i=0,R-1
30        ftt = ftt + (f(i)*f(i))
      CHI = ((R*ftt)/N) - N

      RETURN
      END
```

CPRIMES—C implementation (`cprimes.c`)

```
/* Program that returns the number of prime numbers between 2 and "top" */

int CPRIMES( int top )
{
        int cnt, quotient;
        double tn, td, prime;

        cnt = 1;
        tn = 3;
L9901:  if (tn > top) goto L9904;
            prime = 1;
            td = 3;
L9902:      if ( prime!=1 || td>(tn/2) ) goto L9903;
                quotient = tn / td;
                if ( quotient*td==tn )
                    prime = 0;
                else
                    td = td + 2;

            goto L9902;
L9903:      if ( prime==1 )
                cnt = cnt + 1;

            tn = tn + 2;
            goto L9901;
L9904:  return cnt;
}
```

CPRIMES—Fortran implementation (`cprimes.f`)

```
C/* Program that returns the number of prime numbers between 2 and "top" */

      INTEGER FUNCTION CPRIMES( top )

      INTEGER quotnt
      DOUBLE PRECISION tn, td, prime

      cprimes = 1
      tn = 3.0
9901  IF (tn .GT. top) GOTO 9904
          prime = 1.0
          td = 3.0
9902      IF ( prime .NE. 1.0  .OR.  td .GT. (tn/2.0) ) GOTO 9903
              quotnt = tn / td
              IF ( quotnt*td .EQ. tn ) THEN
                  prime = 0.0
              ELSE
                  td = td + 2.0
              ENDIF
          GOTO 9902
9903      IF ( prime .EQ. 1.0 ) THEN
              cprimes = cprimes + 1.0
          ENDIF
          tn = tn + 2.0
          GOTO 9901
9904  END
```

FIND—C implementation (`find.c`)

```
void FIND (int *A, int N, int F)
/*         INOUT   IN     IN   */
{
       /* F is index into A[].  After execution, all elements to the left of
        * A[F] are less than or equal to A[F] and all elements to the right of
        * A[F] are greater than or equal to A[F].
        * Only the first N elements are considered.
        * From DeMillo, Lipton, and Sayward [DeMillo78apr], based on the
        * program in Hoare's CACM paper on FIND [Hoare71jan].
        *     ASSERT (F.GE.1.AND.F.LE.N.AND.N.GE.1.AND.N.LE.10)
       */

       int M, NS, R, I, J, W;

       M = 1;
       NS = N;
L10:   if (M >= NS) goto L1000;
       R = A[F];
       I = M;
       J = NS;
L20:   if (I >  J) goto L60;
L30:   if (A[I] >= R) goto L40;
       I = I + 1;
       goto L30;
L40:   if (R >= A[J]) goto L50;
       J = J - 1;
       goto L40;
L50:   if (I  > J) goto L20;

       W = A[I];
       A[I] = A[J];
       A[J] = W;
       I = I + 1;
       J = J - 1;
       goto L20;

L60:   if (F >  J) goto L70;
       NS = J;
       goto L10;
L70:   if (I >  F) goto L1000;
       M = I;
       goto L10;
L1000: return;
}
```

FIND—Fortran implementation (find.f)

```
      SUBROUTINE FIND (A, N, F)
      INTEGER A(10), N, F
C             INOUT  IN IN

C     F is index into A().  After execution, all elements to the left of
C     A(F) are less than or equal to A(F) and all elements to the right of
C     A(F) are greater than or equal to A(F).
C     Only the first N elements are considered.
C     From DeMillo, Lipton, and Sayward [DeMillo78apr], based on the
C     program in Hoare's CACM paper on FIND [Hoare71jan].
      ASSERT (F.GE.1.AND.F.LE.N.AND.N.GE.1.AND.N.LE.10)

      INTEGER M, NS, R, I, J, W

      M = 1
      NS = N
10    IF (M.GE.NS) GOTO 1000
      R = A(F)
      I = M
      J = NS
20    IF (I.GT.J) GOTO 60
30    IF (A(I).GE.R) GOTO 40
      I = I + 1
      GOTO 30
40    IF (R.GE.A(J)) GOTO 50
      J = J - 1
      GOTO 40
50    IF (I.GT.J) GOTO 20

      W = A(I)
      A(I) = A(J)
      A(J) = W
      I = I + 1
      J = J - 1
      GOTO 20

60    IF (F.GT.J) GOTO 70
      NS = J
      GOTO 10
70    IF (I.GT.F) GOTO 1000
      M = I
      GOTO 10
1000  RETURN
      END
```

ICHI—C implementation (`ichi.c`)

```
/*
 * The random number generator and the code for calculating
 * the chi-square test come from Robert Sedgewick's "Algorithms (2nd ed)",
 * pp. 511-519, Addison-Wesley.
 *
 * INTEGER version.
 */

#define R 100
              /* frequency tallies for chi-square calculation */
              /*      V                                        */
int  ICHI(int N, int *f)
{
    int t, p, p1, p0, val1, val2, val3, mod1, mod2, i;
    int ftt;

    /* Initialize tallies to zero */
    for (i=0; i<=R-1; i++)
        f[i] = 0;

    p = 1234567;
    for (i=1; i<=N; i++)
    {
        /* Generate a random number, t, using the Linear Congruential
         * Method.  The random number p is calculated each time through
         * the following code.  It is mapped to the range 0 through R-1
         * to produce the random number t.                            */
        p1 = p / 10000;
        p0 = p - (p1 * 10000);

        val1 = (p0 * 3141) + (p1 * 5821);
        mod1 = val1 - ((val1 / 10000) * 10000);

        val2 = (mod1 * 10000) + (p0 * 5821);
        mod2 = val2 - ((val2 / 100000000) * 100000000);

        val3 = mod2 + 1;
        p =    val3 - ((val3 / 100000000) * 100000000);

        /* Adjust to range 0 through R-1. */
        t = ((p / 10000)*R) / 10000;


        /* Tally number of times each random number is produced. */
        f[t] = f[t] + 1;
    }

    /* Perform a Chi-square test on the distribution of random
     * numbers produced by the scheme above.  If the chisquare
     * value is close to "R", then the numbers are "random".     */
    ftt = 0;
    for (i=0; i<=R-1; i++)
        ftt = ftt + (f[i]*f[i]);
    ftt = ((R*ftt)/N) - N;

    return ftt;
}
```

ICHI—Fortran implementation (ichi.f)

```
C*
C* The random number generator and the code for calculating
C* the chi-square test come from Robert Sedgewick's "Algorithms (2nd ed)",
C* pp. 511-519, Addison-Wesley.
C*
      INTEGER FUNCTION ICHI(N)
      INTEGER N
C
      PARAMETER (R=100)
C
      INTEGER t, p, p1, p0, val1, val2, val3, mod1, mod2, I
C     INTEGER f(0:R)
      INTEGER f(0:100)
C          * frequency tallies for chi-square calculation
      INTEGER ftt

C     * Initialize tallies to zero
      DO 10 i=0,R-1
10        f(i) = 0

      p = 1234567
      DO 20 i=1,N
C
C         * Generate a random number, t, using the Linear Congruential
C         * Method.  The random number p is calculated each time through
C         * the following code.  It is mapped to the range 0 through R-1
C         * to produce the random number t.
          p1 = p / 10000
          p0 = p - (p1 * 10000)

          val1 = (p0 * 3141) + (p1 * 5821)
          mod1 = val1 - ((val1 / 10000) * 10000)

          val2 = (mod1 * 10000) + (p0 * 5821)
          mod2 = val2 - ((val2 / 100000000) * 100000000)

          val3 = mod2 + 1
          p =    val3 - ((val3 / 100000000) * 100000000)

C         * Adjust to range 0 through R-1. */
          t = ((p / 10000)*R) / 10000


C         * Tally number of times each random number is produced.
          f(t) = f(t) + 1
20        CONTINUE

C     * Perform a Chi-square test on the distribution of random
C     * numbers produced by the scheme above.  If the chisquare
C     * value is close to "R", then the numbers are "random".
      ftt = 0
      DO 30 i=0,R-1
30        ftt = ftt + (f(i)*f(i))
      ICHI = ((R*ftt)/N) - N

      RETURN
      END
```

ICPRIMES—C implementation (icprimes.c)

```
/* Program that returns the number of prime numbers between 2 and "top" */

int ICPRIMES( int top )
{
        int cnt, tn, td, prime;

        cnt = 1;
        tn = 3;
L9901:  if (tn > top) goto L9904;
            prime = 1;
            td = 3;
L9902:      if ( prime!=1 || td>(tn/2) ) goto L9903;
                if ( tn%td==0 )
                    prime = 0;
                else
                    td = td + 2;

                goto L9902;
L9903:          if ( prime==1 )
                    cnt = cnt + 1;

                tn = tn + 2;
                goto L9901;
L9904:  return cnt;
}
```

ICPRIMES—Fortran implementation (icprimes.f)

```
C/* Program that returns the number of prime numbers between 2 and "top" */

      FUNCTION ICPRIMES( top )

      INTEGER top, tn, td, prime

      icprimes = 1
      tn = 3
9901  IF (tn .GT. top) GOTO 9904
          prime = 1
          td = 3
9902      IF ( prime .NE. 1  .OR.  td .GT. (tn/2) ) GOTO 9903
              IF ( MOD(tn,td) .EQ. 0 ) THEN
                  prime = 0
              ELSE
                  td = td + 2
              ENDIF
          GOTO 9902
9903      IF ( prime .EQ. 1 ) THEN
              icprimes = icprimes + 1
          ENDIF
          tn = tn + 2
          GOTO 9901
9904  END
```

LFIBO—C implementation (`lfibo.c`)

```
/*
  REMARKS.  Leonardo of Pisa, who is also called Leonardo Fibonacci,
  originated the following sequence of numbers in the year 1202:
  0, 1, 1, 2, 3, 5, 8, 13, 21,...  In this sequence, each number is
  the sum of the preceding two and is denoted by F
  (F for Fibonacci and n for number).             n

  Formally, this sequence is defined as

                     F    = 0
                      0

                     F    = 1
                      1

                     F    = F    + F     where n > 1
                      n+2    n+1    n

  The FIBO routine returns the Nth fibonacci number, as defined above.
  An illustrative program using integer numbers.

              in       out                                         */
void FIBO( int N, int *NUM)
{
      int SP[101], FP[101];
      int I, K, NN, SN, FN, DSP, DFP, DSUM, CARRY;

/*    Initialize NUM array to "empty" (all -1). */
      I = 1;
/*    WHILE I <= 100 DO */
L101: if (  I <= 100 ) {
          NUM[I] = -1;
          I = I + 1;
          goto L101;
      }

      if ( N <= 1 ) {
/*        The first two fibonacci numbers are the same as N. */
          NUM[1] = N;
      }
      else {
/*        Calculate according to the formula. */
/*          Initialize FP and SP. */
          SP[1] = 0;
          FP[1] = 1;
          I = 2;
/*        WHILE I <= 100 DO */
L102:     if (  I <= 100 ) {
              SP[I] = -1;
              FP[I] = -1;
              I = I + 1;
              goto L102;
          }
```

```
/*          Now calculate the fibonacci number from the previous two. */
          K = 2;
/*        WHILE K <= N DO */
L103:     if (  K <= N ) {
/*
              NUM = SP + FP */
              NN = 1;
              SN = 1;
              FN = 1;
              CARRY = 0;
/*            WHILE SP[SN] != -1  OR  FP[FN] != -1 DO */
L104:         if (( SP[SN] != -1 )||( FP[FN] != -1 ) ) {
                  if ( SP[SN] != -1 ) {
                      DSP = SP[SN];
                      SN = SN + 1;
                  }
                  else {
                      DSP = 0;
                  }
                  if ( FP[FN] != -1 ) {
                      DFP = FP[FN];
                      FN = FN + 1;
                  }
                  else {
                      DFP = 0;
                  }
                  DSUM = DSP + DFP + CARRY;
                  if ( DSUM <= 9 ) {
                      NUM[NN] = DSUM;
                      CARRY = 0;
                  }
                  else {
                      NUM[NN] = DSUM - 10;
                      CARRY = 1;
                  }
                  NN = NN + 1;
                  goto L104;
              }
              if ( CARRY == 1) {
                  NUM[NN] = 1;
              }
/*
              SP  = FP */
              FN = 1;
/*            WHILE FP[FN] != -1 DO */
L105:         if (  FP[FN] != -1 ) {
                  SP[FN] = FP[FN];
                  FN = FN + 1;
                  goto L105;
              }
/*
              FP  = NUM */
              NN = 1;
/*            WHILE NUM[NN] != -1 DO */
L106:         if (  NUM[NN] != -1 ) {
                  FP[NN] = NUM[NN];
                  NN = NN + 1;
                  goto L106;
              }
/* */
              K = K + 1;
              goto L103;
          }
      }
      return;
}
```

LFIBO—Fortran implementation (`lfibo.f`)

```
C
C  REMARKS.  Leonardo of Pisa, who is also called Leonardo Fibonacci,
C  originated the following sequence of numbers in the year 1202:
C  0, 1, 1, 2, 3, 5, 8, 13, 21,...  In this sequence, each number is
C  the sum of the preceding two and is denoted by F
C  (F for Fibonacci and n for number).            n
C
C  Formally, this sequence is defined as
C
C                     F    = 0
C                      0
C
C                     F    = 1
C                      1
C
C                     F    = F    + F     where n > 1
C                      n+2    n+1    n
C
C  The FIBO routine returns the Nth fibonacci number, as defined above.
C  An illustrative program using integer numbers.
C                    in  out
      SUBROUTINE FIBO(N, NUM)

      INTEGER N, NUM(0:100)
      INTEGER SP(0:100), FP(0:100)
      INTEGER I, K, NN, SN, FN, DSP, DFP, DSUM, CARRY

C     Initialize NUM array to "empty" (all -1).
      I = 1
C     WHILE I <= 100 DO
101   IF ( I .LE. 100 ) THEN
          NUM(I) = -1
          I = I + 1
          GOTO 101
      ENDIF

      IF ( N .LE. 1 ) THEN
C         The first two fibonacci numbers are the same as N.
          NUM(1) = N

      ELSE
C         Calculate according to the formula.
C           Initialize FP and SP.
          SP(1) = 0
          FP(1) = 1
          I = 2
C         WHILE I <= 100 DO
102       IF ( I .LE. 100 ) THEN
              SP(I) = -1
              FP(I) = -1
              I = I + 1
              GOTO 102
          ENDIF
```

```
C          Now calculate the fibonacci number from the previous two.
         K = 2
C        WHILE K <= N DO
103      IF ( K .LE. N ) THEN
C
C            NUM = SP + FP
             NN = 1
             SN = 1
             FN = 1
             CARRY = 0
C            WHILE SP(SN) != -1  OR  FP(FN) != -1 DO
104          IF ( ( SP(SN) .NE. -1 )  .OR.  ( FP(FN) .NE. -1 ) ) THEN
                 IF ( SP(SN) .NE. -1 ) THEN
                     DSP = SP(SN)
                     SN = SN + 1

                 ELSE
                     DSP = 0
                 ENDIF
                 IF ( FP(FN) .NE. -1 ) THEN
                     DFP = FP(FN)
                     FN = FN + 1

                 ELSE
                     DFP = 0
                 ENDIF
                 DSUM = DSP + DFP + CARRY
                 IF ( DSUM .LE. 9 ) THEN
                     NUM(NN) = DSUM
                     CARRY = 0

                 ELSE
                     NUM(NN) = DSUM - 10
                     CARRY = 1
                 ENDIF
                 NN = NN + 1
                 GOTO 104
             ENDIF
             IF ( CARRY .EQ. 1) THEN
                 NUM(NN) = 1
             ENDIF
C
C            SP  = FP
             FN = 1
C            WHILE FP(FN) != -1 DO
105          IF ( FP(FN) .NE. -1 ) THEN
                 SP(FN) = FP(FN)
                 FN = FN + 1
                 GOTO 105
             ENDIF
C
C            FP  = NUM
             NN = 1
C            WHILE NUM(NN) != -1 DO
106          IF ( NUM(NN) .NE. -1 ) THEN
                 FP(NN) = NUM(NN)
                 NN = NN + 1
                 GOTO 106
             ENDIF
C
             K = K + 1
             GOTO 103
         ENDIF
     ENDIF
     RETURN
     END
```

SUMSQRT—C implementation (`sumsqrt.c`)

```
/* Calculates the sum of the square roots of 1...N */

void SUMSQRT( float N, float *SUM )
{
    float NUMBER, SQRT, GUESS, DELTA, EPS;

    EPS = 0.001;
    *SUM = 0.0;

    NUMBER = 1.0;
    while (NUMBER <= N)
    {
        GUESS = NUMBER / 2.0 + 1.0;
        SQRT = 0.0;
        DELTA = GUESS - SQRT;
        while (DELTA > EPS)
        {
            SQRT = GUESS;
            GUESS = (SQRT + NUMBER / SQRT) / 2.0;
            DELTA = GUESS - SQRT;
            if (DELTA < 0.0)
                DELTA =  -DELTA;
        }
        *SUM = *SUM + SQRT;
        NUMBER = NUMBER + 1.0;
    }
}
```

SUMSQRT—Fortran implementation (`sumsqrt.f`)

```
C  Calculates the sum of the square roots of 1...N

      SUBROUTINE SUMSQRT(N, SUM)

      REAL N, SUM, NUMBER, SQRT, GUESS, DELTA, EPS

      EPS = 0.001
      SUM = 0.0

      NUMBER = 1.0
C     while (NUMBER <= N)
10    IF (NUMBER .LE. N) THEN
          GUESS = NUMBER / 2.0 + 1.0
          SQRT = 0.0
          DELTA = GUESS - SQRT
C         while (DELTA > EPS)
20        IF ( DELTA .GT. EPS ) THEN
              SQRT = GUESS
              GUESS = (SQRT + NUMBER / SQRT) / 2.0
              DELTA = GUESS - SQRT
              IF ( DELTA .LT. 0.0 ) THEN
                  DELTA =  -DELTA
              ENDIF
              GOTO 20
          ENDIF
          SUM = SUM + SQRT
          NUMBER = NUMBER + 1.0
          GOTO 10
      ENDIF
      RETURN
      END
```

TRITYP—C implementation (`trityp.c`)

```
void TRITYP(int *I, int *J, int *K, int *TRIANG)
{          /*    all parameters INOUT              */
/*    MATCH IS OUTPUT FROM THE ROUTINE:
         TRIANG = 1 IF TRIANGLE IS SCALENE
         TRIANG = 2 IF TRIANGLE IS ISOSCELES
         TRIANG = 3 IF TRIANGLE IS EQUILATERAL
         TRIANG = 4 IF NOT A TRIANGLE
*/
#define OR  ||
#define AND &&
/*    After a quick confirmation that it's a legal
      triangle, detect any sides of equal length  */

    if (*I<=0 OR *J<=0 OR *K<=0) {
        *TRIANG=4;
        return;
    }
    *TRIANG=0;
    if (*I==*J) *TRIANG=*TRIANG+1;
    if (*I==*K) *TRIANG=*TRIANG+2;
    if (*J==*K) *TRIANG=*TRIANG+3;
    if (*TRIANG==0) {

/*  Confirm it's a legal triangle before declaring
    it to be scalene  */

        if (*I+*J<=*K OR *J+*K<=*I OR *I+*K<=*J)
            *TRIANG = 4;
        else
            *TRIANG = 1;

        return;
    }

/*    Confirm it's a legal triangle before declaring
      it to be isosceles or equilateral  */

    if (*TRIANG>3)
        *TRIANG = 3;
    else if (*TRIANG==1 AND *I+*J>*K)
        *TRIANG = 2;
    else if (*TRIANG==2 AND *I+*K>*J)
        *TRIANG = 2;
    else if (*TRIANG==3 && *J+*K>*I)
        *TRIANG = 2;
    else
        *TRIANG = 4;

    return;
}
```

TRITYP—Fortran implementation (`trityp.f`)

```
      SUBROUTINE TRITYP(I,J,K,TRIANG)
C                       all parameters INOUT
      INTEGER I,J,K,TRIANG

C     MATCH IS OUTPUT FROM THE ROUTINE:
C        TRIANG = 1 IF TRIANGLE IS SCALENE
C        TRIANG = 2 IF TRIANGLE IS ISOSCELES
C        TRIANG = 3 IF TRIANGLE IS EQUILATERAL
C        TRIANG = 4 IF NOT A TRIANGLE

C     After a quick confirmation that it's a legal
C     triangle, detect any sides of equal length

      IF (I.LE.0.OR.J.LE.0.OR.K.LE.0) THEN
          TRIANG=4
          RETURN
      ENDIF
      TRIANG=0
      IF (I.EQ.J) TRIANG=TRIANG+1
      IF (I.EQ.K) TRIANG=TRIANG+2
      IF (J.EQ.K) TRIANG=TRIANG+3
      IF (TRIANG.EQ.0) THEN

C     Confirm it's a legal triangle before declaring
C     it to be scalene

          IF (I+J.LE.K.OR.J+K.LE.I.OR.I+K.LE.J) THEN
              TRIANG = 4
          ELSE
              TRIANG = 1
          ENDIF
          RETURN
      ENDIF

C     Confirm it's a legal triangle before declaring
C     it to be isosceles or equilateral

      IF (TRIANG.GT.3) THEN
          TRIANG = 3
      ELSE IF (TRIANG.EQ.1.AND.I+J.GT.K) THEN
          TRIANG = 2
      ELSE IF (TRIANG.EQ.2.AND.I+K.GT.J) THEN
          TRIANG = 2
      ELSE IF (TRIANG.EQ.3.AND.J+K.GT.I) THEN
          TRIANG = 2
      ELSE
          TRIANG = 4
      ENDIF

      END
```

Appendix E

The Timer Program

```
/* timer.c - A "stopwatch" program that invokes a program and times it. */
/* === $Revision: 1.3 $ === */

#define MAIN

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <sys/wait.h>

#include <sys/time.h>
#include <sys/resource.h>
#ifdef RUSAGE_CHILDREN
  int   setitimer(int which, struct itimerval *value, struct itimerval *ovalue);
  int   getrusage(int who, struct rusage *rusage);
#endif
#ifndef RUSAGE_CHILDREN
  #include <sys/rusage.h>
#endif
#include <sys/times.h>

#define CLK_TCK sysconf(_SC_CLK_TCK)   /* Ticks per second (clock_t)     */
                                       /* _SC_CLK_TCK (=3) from unistd.h */

extern int errno;
extern int sys_nerr;


/* --- The TIMER Routine ---------------------------------------------------- */
void main( int argc, char *argv[] )
{
    char           *exe_args[25];             /* exec argument vector*/
    int            arg;
    int            pid;
    int            return_status;
    int            Result;

    struct rusage  ChildTime0_, ChildTime1_;
    long           ChildUserTime_;            /* in milliseconds           */
    long           ChildSystemTime_;          /* in milliseconds           */
    long           TotalElapsedTime_;         /* in milliseconds           */

    struct tms     Tick0_,  Tick1_;
    clock_t        UserElapsedTicks_;
    clock_t        SystemElapsedTicks_;
    clock_t        TotalElapsedTicks_;


    /* Initial bookkeeping. */
    if (times(&Tick0_)<0)
        (void) fprintf(stderr, "Unable to get initial clock tick\n");
    if (getrusage(RUSAGE_CHILDREN, &ChildTime0_)<0)
        (void) fprintf(stderr, "Unable to get initial RUSAGE_CHILDREN\n");

    /*
     * Set up argument list for invocation of the program to be timed.
     * Note that the following assignments merely assign to the
     * exe_args array the addresses where the argument values
     * will be found and NOT the actual argument values themselves.
     */
    for (arg=1; arg<=argc; arg++)       /* Should copy over NULL at end */
        exe_args[arg-1] = argv[arg];
```

```
    /* Invoke the requested routine. */
    if ( (pid = fork()) < 0)
    {
        fprintf(stderr,"Can't fork to run %s.", exe_args[0]);
        exit(errno);
    }
    else
    {
        if (pid == 0)
        {   /* Child process:  run specified program. */
            execvp(exe_args[0],exe_args);
            fprintf(stderr,"Invocation of %s failed.", exe_args[0]);
            _exit( 1 );
        }
        else
        {   /* Parent process: wait for child to finish. */
            waitpid( pid, &return_status, 0 );
            Result = WIFEXITED(return_status);
        }
    }

    /* Finalizations */
    if (times(&Tick1_)<0)
        (void) fprintf(stderr, "Unable to get final clock tick\n");
    if (getrusage(RUSAGE_CHILDREN, &ChildTime1_)<0)
        (void) fprintf(stderr, "Unable to get final RUSAGE_CHILDREN\n");

    UserElapsedTicks_   = Tick1_.tms_cutime - Tick0_.tms_cutime;
    SystemElapsedTicks_ = Tick1_.tms_cstime - Tick0_.tms_cstime;
    TotalElapsedTicks_  = UserElapsedTicks_ + SystemElapsedTicks_;
    (void) fprintf(stderr,
      "times:  Clock ticks per second (CLK_TCK) = %ld\n",
      CLK_TCK);
    (void) fprintf(stderr,
      "%9ld %9ld %9ld (user+system+total clock ticks)\n",
      UserElapsedTicks_, SystemElapsedTicks_, TotalElapsedTicks_);
    (void) fprintf(stderr,
      "%9ld %9ld %9ld (user+system=total time in milliseconds per ticks)\n",
      (UserElapsedTicks_   * 1000) / CLK_TCK,
      (SystemElapsedTicks_ * 1000) / CLK_TCK,
      (TotalElapsedTicks_  * 1000) / CLK_TCK );

    ChildUserTime_ =
      (1000.0*ChildTime1_.ru_utime.tv_sec + .001*ChildTime1_.ru_utime.tv_usec)
    -
      (1000.0*ChildTime0_.ru_utime.tv_sec + .001*ChildTime0_.ru_utime.tv_usec);

    ChildSystemTime_ =
      (1000.0*ChildTime1_.ru_stime.tv_sec + .001*ChildTime1_.ru_stime.tv_usec)
    -
      (1000.0*ChildTime0_.ru_stime.tv_sec + .001*ChildTime0_.ru_stime.tv_usec);

    TotalElapsedTime_ = ChildUserTime_ + ChildSystemTime_;
    TotalElapsedTime_ = (TotalElapsedTime_ < 10)? 10 : TotalElapsedTime_;

    (void) fprintf(stderr,
      "getrusage: \n");
    (void) fprintf(stderr,
      "%9ld %9ld %9ld (user+system=total time in milliseconds)\n\n",
      ChildUserTime_, ChildSystemTime_, TotalElapsedTime_);

    exit(0);
}
```

Appendix F

The Test Sets

Test Set	Program	Test Case	Test Case Values
A	CHI	1	N = 25000
B	CPRIMES	1	top = 2000
C	FIND	1	A[1:10] = -19 34 0 -4 22 12 222 -57 17 0 N = 9 F = 5
		2	A[1:10] = 7 9 7 0 0 0 0 0 0 0 N = 3 F = 3
		3	A[1:10] = 2 3 1 0 0 0 0 0 0 0 N = 4 F = 3
		4	A[1:10] = -5 -5 -5 -5 0 0 0 0 0 0 N = 4 F = 1
		5	A[1:10] = 1 3 2 0 0 0 0 0 0 0 N = 4 F = 3
		6	A[1:10] = 0 2 3 1 0 0 0 0 0 0 N = 4 F = 3
		7	A[1:10] = 0 0 0 0 0 0 0 0 0 0 N = 1 F = 1
D	ICHI	1	N = 25000
E	ICPRIMES	1	top = 700
F	LFIBO	1	N = 0
		2	N = 1
		3	N = 2
		4	N = 9
		5	N = 10
		6	N = 100
		7	N = 101
G	SUMSQRT	1	N = 250
		2	N = 100
		3	N = 8500

Test Set	Program	Test Case	Test Case Values
H	TRITYP	1	`I =  3    J =  3    K =  3`
		2	`I =  4    J =  4    K =  3`
		3	`I =  4    J =  3    K =  4`
		4	`I =  4    J =  3    K =  3`
		5	`I =  3    J =  4    K =  4`
		6	`I =  3    J =  4    K =  3`
		7	`I =  3    J =  3    K =  4`
		8	`I =  7    J =  4    K =  3`
		9	`I =  2    J =  4    K =  6`
		10	`I =  6    J =  2    K =  4`
		11	`I =  4    J =  6    K =  2`
		12	`I =  1    J =  1    K =  1`
		13	`I =  0    J =  1    K =  1`
		14	`I =  1    J =  0    K =  1`
		15	`I =  1    J =  1    K =  0`
		16	`I =  7    J = 14    K =  7`
		17	`I = 14    J =  7    K =  7`
		18	`I =  7    J =  7    K = 14`
		19	`I = -1    J =  1    K =  1`
		20	`I =  1    J = -1    K =  1`
		21	`I =  1    J =  1    K = -1`
		22	`I =  2    J =  4    K =  7`
		23	`I =  7    J =  2    K =  4`
		24	`I =  4    J =  7    K =  2`
		25	`I =  7    J =  2    K =  8`
		26	`I =  8    J =  7    K =  2`
		27	`I =  2    J =  8    K =  7`
		28	`I = 37    J = 29    K = 29`
		29	`I = 83    J = 83    K = 89`
		30	`I =  2    J = 29    K =  2`
		31	`I = 69    J = 69    K = 69`
		32	`I = 32    J = 32    K = 85`
		33	`I = 38    J = 11    K = 11`
		34	`I = 19    J = 27    K = 19`

Test Set	Program	Test Case	Test Case Values
I	CHI	1	`N = 1000`
		2	`N = 121000`
		3	`N = 245000`
		4	`N = 400000`
		5	`N = 554000`
		6	`N = 3559000`
		7	`N = 6000000`
J	CPRIMES	1	`top = 2000`
		2	`top = 6000`
		3	`top = 8400`
		4	`top = 10700`
		5	`top = 12640`
		6	`top = 33700`
		7	`top = 44200`
K	ICHI	1	`N = 1000`
		2	`N = 121000`
		3	`N = 245000`
		4	`N = 400000`
		5	`N = 554000`
		6	`N = 3559000`
		7	`N = 6000000`
L	ICPRIMES	1	`top = 3000`
		2	`top = 8800`
		3	`top = 12250`
		4	`top = 15550`
		5	`top = 18340`
		6	`top = 49000`
		7	`top = 64150`
M	SUMSQRT	1	`N = 8500`
		2	`N = 58500`
		3	`N = 106700`
		4	`N = 164300`
		5	`N = 221600`
		6	`N = 1283000`
		7	`N = 2070000`
N	CHI	1	`N = 121000`
O	CPRIMES	1	`top = 6000`
P	ICHI	1	`N = 121000`
Q	ICPRIMES	1	`top = 8800`
R	SUMSQRT	1	`N = 58500`

Appendix G

Bounds Study Statistics

This appendix contains the detailed data and statistics from the bounds of performance study described in Chapter IV. The following pages contain the output from the SAS statistics package used in analyzing the study results.

The meanings of the variable identifiers used in the statistical analyses is given below.

- **P:** Program
- **N:** N data parameter (relative to program)
- **W:** Workload (in milliseconds)
- **U:** User Time (in milliseconds)
- **S:** System Time (in milliseconds)
- **CPUTOT:** User+System Time, Total CPU time used (in milliseconds)
- **VNAME:** Version name:
  1. FastTwin (Metamutant running only FastTwin statements)
  2. SlowTwin (Metamutant running only SlowTwin statements)
  3. CX interpreter
  4. Mothra (Rosetta interpreter)

CHI FastTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	chi.c	1000	1050	740	400	1140	FastTwin
2	chi.c	1000	1050	740	400	1140	FastTwin
3	chi.c	1000	1050	770	360	1130	FastTwin
4	chi.c	1000	1050	710	440	1150	FastTwin
5	chi.c	1000	1050	660	440	1100	FastTwin
6	chi.c	1000	1050	780	370	1150	FastTwin
7	chi.c	1000	1050	720	390	1110	FastTwin
8	chi.c	121000	3050	5070	420	5490	FastTwin
9	chi.c	121000	3050	5070	370	5440	FastTwin
10	chi.c	121000	3050	5060	400	5460	FastTwin
11	chi.c	121000	3050	5110	390	5500	FastTwin
12	chi.c	121000	3050	5050	430	5480	FastTwin
13	chi.c	121000	3050	5080	400	5480	FastTwin
14	chi.c	121000	3050	5070	420	5490	FastTwin
15	chi.c	245000	5090	9600	380	9980	FastTwin
16	chi.c	245000	5090	9560	420	9980	FastTwin
17	chi.c	245000	5090	9550	420	9970	FastTwin
18	chi.c	245000	5090	9570	370	9940	FastTwin
19	chi.c	245000	5090	9580	400	9980	FastTwin
20	chi.c	245000	5090	9590	430	10020	FastTwin
21	chi.c	245000	5090	9540	430	9970	FastTwin
22	chi.c	400000	7660	15210	380	15590	FastTwin
23	chi.c	400000	7660	15210	370	15580	FastTwin
24	chi.c	400000	7660	15190	410	15600	FastTwin
25	chi.c	400000	7660	15240	410	15650	FastTwin
26	chi.c	400000	7660	15180	430	15610	FastTwin
27	chi.c	400000	7660	15180	450	15630	FastTwin
28	chi.c	400000	7660	15180	450	15630	FastTwin
29	chi.c	554000	10200	20820	380	21200	FastTwin
30	chi.c	554000	10200	20770	460	21230	FastTwin
31	chi.c	554000	10200	20880	300	21180	FastTwin
32	chi.c	554000	10200	20800	370	21170	FastTwin
33	chi.c	554000	10200	20770	430	21200	FastTwin
34	chi.c	554000	10200	20750	430	21180	FastTwin
35	chi.c	554000	10200	20780	410	21190	FastTwin
36	chi.c	3559000	60080	130620	480	131100	FastTwin
37	chi.c	3559000	60080	129750	460	130210	FastTwin
38	chi.c	3559000	60080	130350	760	131110	FastTwin
39	chi.c	3559000	60080	129820	740	130560	FastTwin
40	chi.c	3559000	60080	129220	560	129780	FastTwin
41	chi.c	3559000	60080	129800	430	130230	FastTwin
42	chi.c	3559000	60080	130310	470	130780	FastTwin
43	chi.c	6000000	100130	218230	450	218680	FastTwin
44	chi.c	6000000	100130	218320	340	218660	FastTwin
45	chi.c	6000000	100130	218240	450	218690	FastTwin
46	chi.c	6000000	100130	218210	430	218640	FastTwin
47	chi.c	6000000	100130	218310	350	218660	FastTwin
48	chi.c	6000000	100130	218260	430	218690	FastTwin
49	chi.c	6000000	100130	218370	740	219110	FastTwin

CHI FastTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	2.97819E+11	2.97819E+11	99999.99	0.0001
Error	47	1.92464E+06	4.09498E+04		
Corrected Total	48	2.97821E+11			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999994	0.351789	202.3605	57523.27

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	2.97819E+11	2.97819E+11	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	2.97819E+11	2.97819E+11	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-1213.826906	-33.54	0.0001	36.19512476
W	2.195662	2696.81	0.0001	0.00081417

CHI SlowTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	chi.c	1000	1050	1380	380	1760	SlowTwin
2	chi.c	1000	1050	1350	390	1740	SlowTwin
3	chi.c	1000	1050	1340	390	1730	SlowTwin
4	chi.c	1000	1050	1320	440	1760	SlowTwin
5	chi.c	1000	1050	1360	400	1760	SlowTwin
6	chi.c	1000	1050	1330	410	1740	SlowTwin
7	chi.c	1000	1050	1360	380	1740	SlowTwin
8	chi.c	121000	3050	77880	450	78330	SlowTwin
9	chi.c	121000	3050	78010	490	78500	SlowTwin
10	chi.c	121000	3050	78050	360	78410	SlowTwin
11	chi.c	121000	3050	78050	430	78480	SlowTwin
12	chi.c	121000	3050	78090	400	78490	SlowTwin
13	chi.c	121000	3050	78020	430	78450	SlowTwin
14	chi.c	121000	3050	78040	400	78440	SlowTwin
15	chi.c	245000	5090	157110	360	157470	SlowTwin
16	chi.c	245000	5090	157060	340	157400	SlowTwin
17	chi.c	245000	5090	156960	450	157410	SlowTwin
18	chi.c	245000	5090	157360	410	157770	SlowTwin
19	chi.c	245000	5090	157000	460	157460	SlowTwin
20	chi.c	245000	5090	157250	470	157720	SlowTwin
21	chi.c	245000	5090	157050	480	157530	SlowTwin
22	chi.c	400000	7660	255930	450	256380	SlowTwin
23	chi.c	400000	7660	256040	450	256490	SlowTwin
24	chi.c	400000	7660	255930	440	256370	SlowTwin
25	chi.c	400000	7660	256380	470	256850	SlowTwin
26	chi.c	400000	7660	256100	410	256510	SlowTwin
27	chi.c	400000	7660	255990	460	256450	SlowTwin
28	chi.c	400000	7660	255920	430	256350	SlowTwin
29	chi.c	554000	10200	354920	450	355370	SlowTwin
30	chi.c	554000	10200	354250	400	354650	SlowTwin
31	chi.c	554000	10200	354300	450	354750	SlowTwin
32	chi.c	554000	10200	354830	460	355290	SlowTwin
33	chi.c	554000	10200	354930	430	355360	SlowTwin
34	chi.c	554000	10200	354430	460	354890	SlowTwin
35	chi.c	554000	10200	354300	410	354710	SlowTwin
36	chi.c	3559000	60080	2276560	3830	2280390	SlowTwin
37	chi.c	3559000	60080	2271560	490	2272050	SlowTwin
38	chi.c	3559000	60080	2271190	510	2271700	SlowTwin
39	chi.c	3559000	60080	2271970	460	2272430	SlowTwin
40	chi.c	3559000	60080	2268010	570	2268580	SlowTwin
41	chi.c	3559000	60080	2271760	830	2272590	SlowTwin
42	chi.c	3559000	60080	2271990	500	2272490	SlowTwin
43	chi.c	6000000	100130	3829150	530	3829680	SlowTwin
44	chi.c	6000000	100130	3836130	550	3836680	SlowTwin
45	chi.c	6000000	100130	3829970	570	3830540	SlowTwin
46	chi.c	6000000	100130	3828230	630	3828860	SlowTwin
47	chi.c	6000000	100130	3829010	520	3829530	SlowTwin
48	chi.c	6000000	100130	3829270	600	3829870	SlowTwin
49	chi.c	6000000	100130	3836720	660	3837380	SlowTwin

CHI SlowTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	9.21428E+13	9.21428E+13	99999.99	0.0001
Error	47	7.26455E+08	1.54565E+07		
Corrected Total	48	9.21436E+13			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999992	0.395754	3931.475	993413.9

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	9.21428E+13	9.21428E+13	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	9.21428E+13	9.21428E+13	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-39744.63750	-56.52	0.0001	703.2014816
W	38.62069	2441.60	0.0001	0.0158177

CHI Mothra
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	chi.f	1000	1050	1900	220	2120	Mothra
2	chi.f	1000	1050	1900	130	2030	Mothra
3	chi.f	1000	1050	1900	170	2070	Mothra
4	chi.f	1000	1050	1900	200	2100	Mothra
5	chi.f	1000	1050	1930	120	2050	Mothra
6	chi.f	1000	1050	1890	190	2080	Mothra
7	chi.f	1000	1050	1900	170	2070	Mothra
8	chi.f	121000	3050	221420	190	221610	Mothra
9	chi.f	121000	3050	221580	190	221770	Mothra
10	chi.f	121000	3050	221550	220	221770	Mothra
11	chi.f	121000	3050	221390	160	221550	Mothra
12	chi.f	121000	3050	221370	190	221560	Mothra
13	chi.f	121000	3050	221380	180	221560	Mothra
14	chi.f	121000	3050	221410	150	221560	Mothra
15	chi.f	245000	5090	448140	130	448270	Mothra
16	chi.f	245000	5090	448450	250	448700	Mothra
17	chi.f	245000	5090	448120	170	448290	Mothra
18	chi.f	245000	5090	448180	180	448360	Mothra
19	chi.f	245000	5090	448140	170	448310	Mothra
20	chi.f	245000	5090	448560	1030	449590	Mothra
21	chi.f	245000	5090	451400	1600	453000	Mothra
22	chi.f	400000	7660	732450	180	732630	Mothra
23	chi.f	400000	7660	732390	230	732620	Mothra
24	chi.f	400000	7660	732460	180	732640	Mothra
25	chi.f	400000	7660	731880	210	732090	Mothra
26	chi.f	400000	7660	732510	160	732670	Mothra
27	chi.f	400000	7660	732450	220	732670	Mothra
28	chi.f	400000	7660	731920	230	732150	Mothra
29	chi.f	554000	10200	1013490	190	1013680	Mothra
30	chi.f	554000	10200	1013850	210	1014060	Mothra
31	chi.f	554000	10200	1014260	230	1014490	Mothra
32	chi.f	554000	10200	1013580	230	1013810	Mothra
33	chi.f	554000	10200	1013580	280	1013860	Mothra
34	chi.f	554000	10200	1013500	260	1013760	Mothra
35	chi.f	554000	10200	1013390	220	1013610	Mothra
36	chi.f	3559000	60080	6509610	490	6510100	Mothra
37	chi.f	3559000	60080	6511390	390	6511780	Mothra
38	chi.f	3559000	60080	6510850	460	6511310	Mothra
39	chi.f	3559000	60080	6536780	1330	6538110	Mothra
40	chi.f	3559000	60080	6508770	450	6509220	Mothra
41	chi.f	3559000	60080	6511810	420	6512230	Mothra
42	chi.f	3559000	60080	6623360	2300	6625660	Mothra
43	chi.f	6000000	100130	10970930	610	10971540	Mothra
44	chi.f	6000000	100130	11008090	1550	11009640	Mothra
45	chi.f	6000000	100130	10975270	680	10975950	Mothra
46	chi.f	6000000	100130	10980060	660	10980720	Mothra
47	chi.f	6000000	100130	11012040	1140	11013180	Mothra
48	chi.f	6000000	100130	10971240	570	10971810	Mothra
49	chi.f	6000000	100130	10976910	590	10977500	Mothra

CHI Mothra
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	7.58657E+14	7.58657E+14	99999.99	0.0001
Error	47	1.40043E+10	2.97964E+08		
Corrected Total	48	7.58671E+14			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999982	0.606088	17261.62	2848039

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	7.58657E+14	7.58657E+14	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	7.58657E+14	7.58657E+14	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-116512.3576	-37.74	0.0001	3087.492117
W	110.8184	1595.66	0.0001	0.069450

CPRIMES FastTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	cprimes.	2000	970	1030	350	1380	FastTwin
2	cprimes.	2000	970	990	430	1420	FastTwin
3	cprimes.	2000	970	1020	350	1370	FastTwin
4	cprimes.	2000	970	1040	340	1380	FastTwin
5	cprimes.	2000	970	1000	390	1390	FastTwin
6	cprimes.	2000	970	1010	400	1410	FastTwin
7	cprimes.	2000	970	1000	400	1400	FastTwin
8	cprimes.	6000	2950	5720	390	6110	FastTwin
9	cprimes.	6000	2950	5640	480	6120	FastTwin
10	cprimes.	6000	2950	5730	380	6110	FastTwin
11	cprimes.	6000	2950	5710	400	6110	FastTwin
12	cprimes.	6000	2950	5650	470	6120	FastTwin
13	cprimes.	6000	2950	5710	380	6090	FastTwin
14	cprimes.	6000	2950	5670	400	6070	FastTwin
15	cprimes.	8400	4970	10450	360	10810	FastTwin
16	cprimes.	8400	4970	10400	420	10820	FastTwin
17	cprimes.	8400	4970	10410	430	10840	FastTwin
18	cprimes.	8400	4970	10410	420	10830	FastTwin
19	cprimes.	8400	4970	10390	410	10800	FastTwin
20	cprimes.	8400	4970	10420	390	10810	FastTwin
21	cprimes.	8400	4970	10410	460	10870	FastTwin
22	cprimes.	10700	7440	16330	370	16700	FastTwin
23	cprimes.	10700	7440	16310	430	16740	FastTwin
24	cprimes.	10700	7440	16290	430	16720	FastTwin
25	cprimes.	10700	7440	16310	420	16730	FastTwin
26	cprimes.	10700	7440	16300	420	16720	FastTwin
27	cprimes.	10700	7440	16310	430	16740	FastTwin
28	cprimes.	10700	7440	16360	390	16750	FastTwin
29	cprimes.	12640	9940	22240	370	22610	FastTwin
30	cprimes.	12640	9940	22250	370	22620	FastTwin
31	cprimes.	12640	9940	22240	360	22600	FastTwin
32	cprimes.	12640	9940	22210	360	22570	FastTwin
33	cprimes.	12640	9940	22270	350	22620	FastTwin
34	cprimes.	12640	9940	22220	360	22580	FastTwin
35	cprimes.	12640	9940	22180	390	22570	FastTwin
36	cprimes.	33700	59990	140100	340	140440	FastTwin
37	cprimes.	33700	59990	140030	390	140420	FastTwin
38	cprimes.	33700	59990	140070	360	140430	FastTwin
39	cprimes.	33700	59990	140070	410	140480	FastTwin
40	cprimes.	33700	59990	140090	380	140470	FastTwin
41	cprimes.	33700	59990	140140	310	140450	FastTwin
42	cprimes.	33700	59990	140010	410	140420	FastTwin
43	cprimes.	44200	99910	233920	420	234340	FastTwin
44	cprimes.	44200	99910	233970	410	234380	FastTwin
45	cprimes.	44200	99910	233930	440	234370	FastTwin
46	cprimes.	44200	99910	233910	470	234380	FastTwin
47	cprimes.	44200	99910	233970	420	234390	FastTwin
48	cprimes.	44200	99910	233930	410	234340	FastTwin
49	cprimes.	44200	99910	233930	450	234380	FastTwin

CPRIMES FastTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	3.42179E+11	3.42179E+11	99999.99	0.0001
Error	47	8.66556E+04	1.84374E+03		
Corrected Total	48	3.42179E+11			

R-Square	C.V.	Root MSE	CPUTOT Mean
1.000000	0.069503	42.93876	61780.00

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	3.42179E+11	3.42179E+11	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	3.42179E+11	3.42179E+11	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-838.7392359	-109.42	0.0001	7.66518877
W	2.3544673	13623.13	0.0001	0.00017283

CPRIMES SlowTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	cprimes.	2000	970	13670	340	14010	SlowTwin
2	cprimes.	2000	970	13670	350	14020	SlowTwin
3	cprimes.	2000	970	13660	380	14040	SlowTwin
4	cprimes.	2000	970	13630	360	13990	SlowTwin
5	cprimes.	2000	970	13620	390	14010	SlowTwin
6	cprimes.	2000	970	13640	380	14020	SlowTwin
7	cprimes.	2000	970	13690	340	14030	SlowTwin
8	cprimes.	6000	2950	105800	960	106760	SlowTwin
9	cprimes.	6000	2950	105530	460	105990	SlowTwin
10	cprimes.	6000	2950	105580	400	105980	SlowTwin
11	cprimes.	6000	2950	105600	410	106010	SlowTwin
12	cprimes.	6000	2950	105850	430	106280	SlowTwin
13	cprimes.	6000	2950	105420	420	105840	SlowTwin
14	cprimes.	6000	2950	105570	460	106030	SlowTwin
15	cprimes.	8400	4970	198400	410	198810	SlowTwin
16	cprimes.	8400	4970	197930	430	198360	SlowTwin
17	cprimes.	8400	4970	197960	440	198400	SlowTwin
18	cprimes.	8400	4970	197920	420	198340	SlowTwin
19	cprimes.	8400	4970	197980	430	198410	SlowTwin
20	cprimes.	8400	4970	197710	480	198190	SlowTwin
21	cprimes.	8400	4970	197870	460	198330	SlowTwin
22	cprimes.	10700	7440	305740	480	306220	SlowTwin
23	cprimes.	10700	7440	306180	390	306570	SlowTwin
24	cprimes.	10700	7440	305550	450	306000	SlowTwin
25	cprimes.	10700	7440	306150	370	306520	SlowTwin
26	cprimes.	10700	7440	305860	370	306230	SlowTwin
27	cprimes.	10700	7440	305550	380	305930	SlowTwin
28	cprimes.	10700	7440	305880	370	306250	SlowTwin
29	cprimes.	12640	9940	419890	1740	421630	SlowTwin
30	cprimes.	12640	9940	418270	420	418690	SlowTwin
31	cprimes.	12640	9940	417770	410	418180	SlowTwin
32	cprimes.	12640	9940	417590	430	418020	SlowTwin
33	cprimes.	12640	9940	418560	440	419000	SlowTwin
34	cprimes.	12640	9940	417310	410	417720	SlowTwin
35	cprimes.	12640	9940	417550	440	417990	SlowTwin
36	cprimes.	33700	59990	2669910	600	2670510	SlowTwin
37	cprimes.	33700	59990	2669950	580	2670530	SlowTwin
38	cprimes.	33700	59990	2665880	4830	2670710	SlowTwin
39	cprimes.	33700	59990	2669780	4220	2674000	SlowTwin
40	cprimes.	33700	59990	2664680	540	2665220	SlowTwin
41	cprimes.	33700	59990	2667320	600	2667920	SlowTwin
42	cprimes.	33700	59990	2664740	600	2665340	SlowTwin
43	cprimes.	44200	99910	4602830	670	4603500	SlowTwin
44	cprimes.	44200	99910	4595340	730	4596070	SlowTwin
45	cprimes.	44200	99910	4593580	1400	4594980	SlowTwin
46	cprimes.	44200	99910	4594050	1050	4595100	SlowTwin
47	cprimes.	44200	99910	4602230	620	4602850	SlowTwin
48	cprimes.	44200	99910	4593670	1370	4595040	SlowTwin
49	cprimes.	44200	99910	4593940	620	4594560	SlowTwin

CPRIMES SlowTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	1.30977E+14	1.30977E+14	99999.99	0.0001
Error	47	3.12561E+10	6.65024E+08		
Corrected Total	48	1.31008E+14			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999761	2.172238	25788.07	1187166

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	1.30977E+14	1.30977E+14	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	1.30977E+14	1.30977E+14	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-37943.77978	-8.24	0.0001	4603.542478
W	46.06418	443.79	0.0001	0.103797

CPRIMES CX
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	cprimes.	2000	970	7300	490	7790	CX
2	cprimes.	2000	970	7270	360	7630	CX
3	cprimes.	2000	970	7300	330	7630	CX
4	cprimes.	2000	970	7290	300	7590	CX
5	cprimes.	2000	970	7280	380	7660	CX
6	cprimes.	2000	970	7270	340	7610	CX
7	cprimes.	2000	970	7290	360	7650	CX
8	cprimes.	6000	2950	55230	540	55770	CX
9	cprimes.	6000	2950	55230	380	55610	CX
10	cprimes.	6000	2950	55190	400	55590	CX
11	cprimes.	6000	2950	55230	390	55620	CX
12	cprimes.	6000	2950	55250	370	55620	CX
13	cprimes.	6000	2950	55210	320	55530	CX
14	cprimes.	6000	2950	55180	320	55500	CX
15	cprimes.	8400	4970	103550	500	104050	CX
16	cprimes.	8400	4970	103550	370	103920	CX
17	cprimes.	8400	4970	103570	300	103870	CX
18	cprimes.	8400	4970	103630	360	103990	CX
19	cprimes.	8400	4970	103590	360	103950	CX
20	cprimes.	8400	4970	103500	330	103830	CX
21	cprimes.	8400	4970	103510	310	103820	CX
22	cprimes.	10700	7440	164050	410	164460	CX
23	cprimes.	10700	7440	163900	450	164350	CX
24	cprimes.	10700	7440	164060	380	164440	CX
25	cprimes.	10700	7440	164040	410	164450	CX
26	cprimes.	10700	7440	163980	400	164380	CX
27	cprimes.	10700	7440	164120	330	164450	CX
28	cprimes.	10700	7440	164000	300	164300	CX
29	cprimes.	12640	9940	224170	390	224560	CX
30	cprimes.	12640	9940	224050	320	224370	CX
31	cprimes.	12640	9940	224170	330	224500	CX
32	cprimes.	12640	9940	224090	410	224500	CX
33	cprimes.	12640	9940	224110	410	224520	CX
34	cprimes.	12640	9940	224220	330	224550	CX
35	cprimes.	12640	9940	223910	380	224290	CX
36	cprimes.	33700	59990	1431710	460	1432170	CX
37	cprimes.	33700	59990	1431520	480	1432000	CX
38	cprimes.	33700	59990	1431560	440	1432000	CX
39	cprimes.	33700	59990	1431850	400	1432250	CX
40	cprimes.	33700	59990	1431330	520	1431850	CX
41	cprimes.	33700	59990	1431520	480	1432000	CX
42	cprimes.	33700	59990	1431620	510	1432130	CX
43	cprimes.	44200	99910	2393160	600	2393760	CX
44	cprimes.	44200	99910	2392940	500	2393440	CX
45	cprimes.	44200	99910	2397870	800	2398670	CX
46	cprimes.	44200	99910	2404690	940	2405630	CX
47	cprimes.	44200	99910	2403780	310	2404090	CX
48	cprimes.	44200	99910	2404110	450	2404560	CX
49	cprimes.	44200	99910	2402280	460	2402740	CX

CPRIMES CX
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	3.60698E+13	3.60698E+13	99999.99	0.0001
Error	47	2.05122E+08	4.36430E+06		
Corrected Total	48	3.60700E+13			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999994	0.333225	2089.090	626931.4

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	3.60698E+13	3.60698E+13	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	3.60698E+13	3.60698E+13	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-15977.93539	-42.84	0.0001	372.9328115
W	24.17342	2874.85	0.0001	0.0084086

CPRIMES Mothra
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	cprimes.	2000	970	44070	190	44260	Mothra
2	cprimes.	2000	970	44100	160	44260	Mothra
3	cprimes.	2000	970	44060	160	44220	Mothra
4	cprimes.	2000	970	44080	130	44210	Mothra
5	cprimes.	2000	970	44070	210	44280	Mothra
6	cprimes.	2000	970	44060	200	44260	Mothra
7	cprimes.	2000	970	44080	200	44280	Mothra
8	cprimes.	6000	2950	338730	200	338930	Mothra
9	cprimes.	6000	2950	338710	170	338880	Mothra
10	cprimes.	6000	2950	338730	250	338980	Mothra
11	cprimes.	6000	2950	338720	140	338860	Mothra
12	cprimes.	6000	2950	338730	200	338930	Mothra
13	cprimes.	6000	2950	339040	570	339610	Mothra
14	cprimes.	6000	2950	338990	350	339340	Mothra
15	cprimes.	8400	4850	635850	790	636640	Mothra
16	cprimes.	8400	4850	635860	240	636100	Mothra
17	cprimes.	8400	4850	637080	1040	638120	Mothra
18	cprimes.	8400	4850	635710	250	635960	Mothra
19	cprimes.	8400	4850	635700	220	635920	Mothra
20	cprimes.	8400	4850	635820	370	636190	Mothra
21	cprimes.	8400	4850	635610	160	635770	Mothra
22	cprimes.	10700	7440	1008560	170	1008730	Mothra
23	cprimes.	10700	7440	1008610	260	1008870	Mothra
24	cprimes.	10700	7440	1008500	270	1008770	Mothra
25	cprimes.	10700	7440	1008320	210	1008530	Mothra
26	cprimes.	10700	7440	1008380	250	1008630	Mothra
27	cprimes.	10700	7440	1008630	170	1008800	Mothra
28	cprimes.	10700	7440	1008700	240	1008940	Mothra
29	cprimes.	12640	9940	1378010	310	1378320	Mothra
30	cprimes.	12640	9940	1377660	250	1377910	Mothra
31	cprimes.	12640	9940	1378120	180	1378300	Mothra
32	cprimes.	12640	9940	1378090	230	1378320	Mothra
33	cprimes.	12640	9940	1377790	190	1377980	Mothra
34	cprimes.	12640	9940	1377890	200	1378090	Mothra
35	cprimes.	12640	9940	1378510	210	1378720	Mothra
36	cprimes.	33700	59990	8798430	770	8799200	Mothra
37	cprimes.	33700	59990	8798680	1580	8800260	Mothra
38	cprimes.	33700	59990	8796990	710	8797700	Mothra
39	cprimes.	33700	59990	8796240	510	8796750	Mothra
40	cprimes.	33700	59990	8797470	1320	8798790	Mothra
41	cprimes.	33700	59990	8817750	800	8818550	Mothra
42	cprimes.	33700	59990	8830010	1560	8831570	Mothra
43	cprimes.	44200	99910	14719900	2350	14722250	Mothra
44	cprimes.	44200	99910	14702710	3450	14706160	Mothra
45	cprimes.	44200	99910	14706870	1410	14708280	Mothra
46	cprimes.	44200	99910	14713630	3190	14716820	Mothra
47	cprimes.	44200	99910	14715820	1440	14717260	Mothra
48	cprimes.	44200	99910	14711310	7350	14718660	Mothra
49	cprimes.	44200	99910	14718030	5040	14723070	Mothra

CPRIMES Mothra
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	1.35768E+15	1.35768E+15	99999.99	0.0001
Error	47	2.90326E+09	6.17716E+07		
Corrected Total	48	1.35768E+15			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999998	0.204303	7859.489	3846984

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	1.35768E+15	1.35768E+15	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	1.35768E+15	1.35768E+15	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-93671.62763	-66.79	0.0001	1402.558507
W	148.26440	4688.18	0.0001	0.031625

ICHI FastTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	ichi.c	1000	1040	680	450	1130	FastTwin
2	ichi.c	1000	1040	700	410	1110	FastTwin
3	ichi.c	1000	1040	740	400	1140	FastTwin
4	ichi.c	1000	1040	670	430	1100	FastTwin
5	ichi.c	1000	1040	740	340	1080	FastTwin
6	ichi.c	1000	1040	820	290	1110	FastTwin
7	ichi.c	1000	1040	670	440	1110	FastTwin
8	ichi.c	121000	3040	5060	410	5470	FastTwin
9	ichi.c	121000	3040	5060	380	5440	FastTwin
10	ichi.c	121000	3040	5030	430	5460	FastTwin
11	ichi.c	121000	3040	5030	440	5470	FastTwin
12	ichi.c	121000	3040	5020	420	5440	FastTwin
13	ichi.c	121000	3040	5080	340	5420	FastTwin
14	ichi.c	121000	3040	5020	470	5490	FastTwin
15	ichi.c	245000	5030	9490	450	9940	FastTwin
16	ichi.c	245000	5030	9490	470	9960	FastTwin
17	ichi.c	245000	5030	9510	440	9950	FastTwin
18	ichi.c	245000	5030	9480	460	9940	FastTwin
19	ichi.c	245000	5030	9510	390	9900	FastTwin
20	ichi.c	245000	5030	9510	390	9900	FastTwin
21	ichi.c	245000	5030	9470	440	9910	FastTwin
22	ichi.c	400000	7540	15050	450	15500	FastTwin
23	ichi.c	400000	7540	15080	410	15490	FastTwin
24	ichi.c	400000	7540	15060	380	15440	FastTwin
25	ichi.c	400000	7540	15070	430	15500	FastTwin
26	ichi.c	400000	7540	15040	440	15480	FastTwin
27	ichi.c	400000	7540	15080	380	15460	FastTwin
28	ichi.c	400000	7540	15060	430	15490	FastTwin
29	ichi.c	554000	9990	20460	460	20920	FastTwin
30	ichi.c	554000	9990	20500	400	20900	FastTwin
31	ichi.c	554000	9990	20430	420	20850	FastTwin
32	ichi.c	554000	9990	20510	400	20910	FastTwin
33	ichi.c	554000	9990	20540	370	20910	FastTwin
34	ichi.c	554000	9990	20520	360	20880	FastTwin
35	ichi.c	554000	9990	20420	500	20920	FastTwin
36	ichi.c	3559000	58840	128580	410	128990	FastTwin
37	ichi.c	3559000	58840	128510	520	129030	FastTwin
38	ichi.c	3559000	58840	128610	410	129020	FastTwin
39	ichi.c	3559000	58840	128480	490	128970	FastTwin
40	ichi.c	3559000	58840	128540	420	128960	FastTwin
41	ichi.c	3559000	58840	128470	530	129000	FastTwin
42	ichi.c	3559000	58840	128460	460	128920	FastTwin
43	ichi.c	6000000	98440	216600	1070	217670	FastTwin
44	ichi.c	6000000	98440	217190	910	218100	FastTwin
45	ichi.c	6000000	98440	216120	850	216970	FastTwin
46	ichi.c	6000000	98440	217000	1030	218030	FastTwin
47	ichi.c	6000000	98440	216670	740	217410	FastTwin
48	ichi.c	6000000	98440	216650	1580	218230	FastTwin
49	ichi.c	6000000	98440	216780	870	217650	FastTwin

ICHI FastTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	2.94256E+11	2.94256E+11	99999.99	0.0001
Error	47	3.33797E+06	7.10207E+04		
Corrected Total	48	2.94259E+11			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999989	0.466858	266.4971	57083.06

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	2.94256E+11	2.94256E+11	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	2.94256E+11	2.94256E+11	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-1303.912826	-27.35	0.0001	47.66757755
W	2.222210	2035.49	0.0001	0.00109173

ICHI SlowTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	ichi.c	1000	1040	1360	370	1730	SlowTwin
2	ichi.c	1000	1040	1330	440	1770	SlowTwin
3	ichi.c	1000	1040	1380	410	1790	SlowTwin
4	ichi.c	1000	1040	1400	380	1780	SlowTwin
5	ichi.c	1000	1040	1350	380	1730	SlowTwin
6	ichi.c	1000	1040	1360	420	1780	SlowTwin
7	ichi.c	1000	1040	1340	400	1740	SlowTwin
8	ichi.c	121000	3040	77840	420	78260	SlowTwin
9	ichi.c	121000	3040	77720	450	78170	SlowTwin
10	ichi.c	121000	3040	77850	410	78260	SlowTwin
11	ichi.c	121000	3040	77800	420	78220	SlowTwin
12	ichi.c	121000	3040	77800	450	78250	SlowTwin
13	ichi.c	121000	3040	77730	420	78150	SlowTwin
14	ichi.c	121000	3040	77810	400	78210	SlowTwin
15	ichi.c	245000	5030	156640	500	157140	SlowTwin
16	ichi.c	245000	5030	156630	430	157060	SlowTwin
17	ichi.c	245000	5030	156690	400	157090	SlowTwin
18	ichi.c	245000	5030	156850	460	157310	SlowTwin
19	ichi.c	245000	5030	156870	450	157320	SlowTwin
20	ichi.c	245000	5030	156590	470	157060	SlowTwin
21	ichi.c	245000	5030	156710	470	157180	SlowTwin
22	ichi.c	400000	7540	255460	430	255890	SlowTwin
23	ichi.c	400000	7540	255400	470	255870	SlowTwin
24	ichi.c	400000	7540	255360	400	255760	SlowTwin
25	ichi.c	400000	7540	255430	430	255860	SlowTwin
26	ichi.c	400000	7540	255350	470	255820	SlowTwin
27	ichi.c	400000	7540	255200	530	255730	SlowTwin
28	ichi.c	400000	7540	255730	450	256180	SlowTwin
29	ichi.c	554000	9990	353130	430	353560	SlowTwin
30	ichi.c	554000	9990	353270	380	353650	SlowTwin
31	ichi.c	554000	9990	353530	480	354010	SlowTwin
32	ichi.c	554000	9990	353370	540	353910	SlowTwin
33	ichi.c	554000	9990	353300	480	353780	SlowTwin
34	ichi.c	554000	9990	353320	510	353830	SlowTwin
35	ichi.c	554000	9990	353290	480	353770	SlowTwin
36	ichi.c	3559000	58840	2265340	770	2266110	SlowTwin
37	ichi.c	3559000	58840	2265790	1290	2267080	SlowTwin
38	ichi.c	3559000	58840	2265750	940	2266690	SlowTwin
39	ichi.c	3559000	58840	2274280	4580	2278860	SlowTwin
40	ichi.c	3559000	58840	2299590	2090	2301680	SlowTwin
41	ichi.c	3559000	58840	2265800	960	2266760	SlowTwin
42	ichi.c	3559000	58840	2270270	1040	2271310	SlowTwin
43	ichi.c	6000000	98440	3822140	2670	3824810	SlowTwin
44	ichi.c	6000000	98440	3825580	1000	3826580	SlowTwin
45	ichi.c	6000000	98440	3819380	960	3820340	SlowTwin
46	ichi.c	6000000	98440	3876060	2200	3878260	SlowTwin
47	ichi.c	6000000	98440	3853700	1120	3854820	SlowTwin
48	ichi.c	6000000	98440	3819110	1090	3820200	SlowTwin
49	ichi.c	6000000	98440	3819110	1020	3820130	SlowTwin

ICHI SlowTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	9.23171E+13	9.23171E+13	99999.99	0.0001
Error	47	4.15487E+09	8.84014E+07		
Corrected Total	48	9.23213E+13			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999955	0.946182	9402.202	993699.0

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	9.23171E+13	9.23171E+13	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	9.23171E+13	9.23171E+13	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-40477.76487	-24.07	0.0001	1681.745226
W	39.36079	1021.91	0.0001	0.038517

ICHI CX
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	ichi.c	1000	1040	250	640	890	CX
2	ichi.c	1000	1040	440	400	840	CX
3	ichi.c	1000	1040	440	400	840	CX
4	ichi.c	1000	1040	460	440	900	CX
5	ichi.c	1000	1040	410	410	820	CX
6	ichi.c	1000	1040	420	410	830	CX
7	ichi.c	1000	1040	450	430	880	CX
8	ichi.c	121000	3040	27570	400	27970	CX
9	ichi.c	121000	3040	26930	470	27400	CX
10	ichi.c	121000	3040	26880	750	27630	CX
11	ichi.c	121000	3040	27430	320	27750	CX
12	ichi.c	121000	3040	27440	480	27920	CX
13	ichi.c	121000	3040	27120	1070	28190	CX
14	ichi.c	121000	3040	27590	600	28190	CX
15	ichi.c	245000	5030	54990	610	55600	CX
16	ichi.c	245000	5030	55530	480	56010	CX
17	ichi.c	245000	5030	55020	450	55470	CX
18	ichi.c	245000	5030	55020	340	55360	CX
19	ichi.c	245000	5030	55040	350	55390	CX
20	ichi.c	245000	5030	55010	370	55380	CX
21	ichi.c	245000	5030	54990	370	55360	CX
22	ichi.c	400000	7540	89600	550	90150	CX
23	ichi.c	400000	7540	89630	380	90010	CX
24	ichi.c	400000	7540	89630	350	89980	CX
25	ichi.c	400000	7540	89620	390	90010	CX
26	ichi.c	400000	7540	89640	360	90000	CX
27	ichi.c	400000	7540	89650	340	89990	CX
28	ichi.c	400000	7540	89570	430	90000	CX
29	ichi.c	554000	9990	124060	440	124500	CX
30	ichi.c	554000	9990	124000	410	124410	CX
31	ichi.c	554000	9990	124060	290	124350	CX
32	ichi.c	554000	9990	124010	400	124410	CX
33	ichi.c	554000	9990	123970	460	124430	CX
34	ichi.c	554000	9990	124070	390	124460	CX
35	ichi.c	554000	9990	124060	350	124410	CX
36	ichi.c	3559000	58840	796230	500	796730	CX
37	ichi.c	3559000	58840	796220	490	796710	CX
38	ichi.c	3559000	58840	796950	1260	798210	CX
39	ichi.c	3559000	58840	796330	600	796930	CX
40	ichi.c	3559000	58840	796200	510	796710	CX
41	ichi.c	3559000	58840	796160	470	796630	CX
42	ichi.c	3559000	58840	796460	840	797300	CX
43	ichi.c	6000000	98440	1340740	740	1341480	CX
44	ichi.c	6000000	98440	1342080	1260	1343340	CX
45	ichi.c	6000000	98440	1341290	2240	1343530	CX
46	ichi.c	6000000	98440	1341650	1260	1342910	CX
47	ichi.c	6000000	98440	1341940	1910	1343850	CX
48	ichi.c	6000000	98440	1343200	2700	1345900	CX
49	ichi.c	6000000	98440	1342530	2520	1345050	CX

ICHI CX
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	1.13272E+13	1.13272E+13	99999.99	0.0001
Error	47	1.99237E+07	4.23908E+05		
Corrected Total	48	1.13273E+13			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999998	0.186829	651.0819	348490.0

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	1.13272E+13	1.13272E+13	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	1.13272E+13	1.13272E+13	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-13765.96322	-118.21	0.0001	116.4571689
W	13.78747	5169.24	0.0001	0.0026672

ICHI Mothra
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	ichi.f	1000	1040	1890	240	2130	Mothra
2	ichi.f	1000	1040	1910	100	2010	Mothra
3	ichi.f	1000	1040	1930	230	2160	Mothra
4	ichi.f	1000	1040	1890	230	2120	Mothra
5	ichi.f	1000	1040	1920	180	2100	Mothra
6	ichi.f	1000	1040	1880	190	2070	Mothra
7	ichi.f	1000	1040	1930	140	2070	Mothra
8	ichi.f	121000	3040	221280	200	221480	Mothra
9	ichi.f	121000	3040	221310	250	221560	Mothra
10	ichi.f	121000	3040	221290	250	221540	Mothra
11	ichi.f	121000	3040	221230	190	221420	Mothra
12	ichi.f	121000	3040	221290	210	221500	Mothra
13	ichi.f	121000	3040	221230	180	221410	Mothra
14	ichi.f	121000	3040	221230	190	221420	Mothra
15	ichi.f	245000	5030	448030	250	448280	Mothra
16	ichi.f	245000	5030	448080	210	448290	Mothra
17	ichi.f	245000	5030	447920	230	448150	Mothra
18	ichi.f	245000	5030	448080	280	448360	Mothra
19	ichi.f	245000	5030	448050	250	448300	Mothra
20	ichi.f	245000	5030	448870	1650	450520	Mothra
21	ichi.f	245000	5030	448720	660	449380	Mothra
22	ichi.f	400000	7540	731340	210	731550	Mothra
23	ichi.f	400000	7540	731780	300	732080	Mothra
24	ichi.f	400000	7540	731070	220	731290	Mothra
25	ichi.f	400000	7540	731320	250	731570	Mothra
26	ichi.f	400000	7540	731510	310	731820	Mothra
27	ichi.f	400000	7540	731480	260	731740	Mothra
28	ichi.f	400000	7540	730060	380	730440	Mothra
29	ichi.f	554000	9990	1012550	330	1012880	Mothra
30	ichi.f	554000	9990	1013740	780	1014520	Mothra
31	ichi.f	554000	9990	1014570	1080	1015650	Mothra
32	ichi.f	554000	9990	1012620	330	1012950	Mothra
33	ichi.f	554000	9990	1012610	290	1012900	Mothra
34	ichi.f	554000	9990	1012500	350	1012850	Mothra
35	ichi.f	554000	9990	1012780	430	1013210	Mothra
36	ichi.f	3559000	58840	6506180	2010	6508190	Mothra
37	ichi.f	3559000	58840	6508280	2690	6510970	Mothra
38	ichi.f	3559000	58840	6511880	3580	6515460	Mothra
39	ichi.f	3559000	58840	6509820	4900	6514720	Mothra
40	ichi.f	3559000	58840	6517000	29320	6546320	Mothra
41	ichi.f	3559000	58840	6519650	16200	6535850	Mothra
42	ichi.f	3559000	58840	6506350	6350	6512700	Mothra
43	ichi.f	6000000	98440	10973480	7200	10980680	Mothra
44	ichi.f	6000000	98440	10993150	61150	11054300	Mothra
45	ichi.f	6000000	98440	10986900	22070	11008970	Mothra
46	ichi.f	6000000	98440	10978600	16650	10995250	Mothra
47	ichi.f	6000000	98440	10999290	22610	11021900	Mothra
48	ichi.f	6000000	98440	11062540	5660	11068200	Mothra
49	ichi.f	6000000	98440	11334320	2730	11337050	Mothra

ICHI Mothra
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	7.67413E+14	7.67413E+14	99999.99	0.0001
Error	47	1.03080E+11	2.19320E+09		
Corrected Total	48	7.67516E+14			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999866	1.638727	46831.58	2857802

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	7.67413E+14	7.67413E+14	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	7.67413E+14	7.67413E+14	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-123927.9695	-14.79	0.0001	8376.632159
W	113.4847	591.53	0.0001	0.191850

ICPRIMES FastTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	icprimes	3000	980	1370	320	1690	FastTwin
2	icprimes	3000	980	1320	410	1730	FastTwin
3	icprimes	3000	980	1180	510	1690	FastTwin
4	icprimes	3000	980	1270	450	1720	FastTwin
5	icprimes	3000	980	1220	490	1710	FastTwin
6	icprimes	3000	980	1190	500	1690	FastTwin
7	icprimes	3000	980	1270	420	1690	FastTwin
8	icprimes	8800	2980	7710	500	8210	FastTwin
9	icprimes	8800	2980	7730	440	8170	FastTwin
10	icprimes	8800	2980	7730	460	8190	FastTwin
11	icprimes	8800	2980	7770	400	8170	FastTwin
12	icprimes	8800	2980	7770	420	8190	FastTwin
13	icprimes	8800	2980	7660	520	8180	FastTwin
14	icprimes	8800	2980	7730	420	8150	FastTwin
15	icprimes	12250	4940	14160	390	14550	FastTwin
16	icprimes	12250	4940	14140	410	14550	FastTwin
17	icprimes	12250	4940	14200	310	14510	FastTwin
18	icprimes	12250	4940	14120	390	14510	FastTwin
19	icprimes	12250	4940	14190	360	14550	FastTwin
20	icprimes	12250	4940	14170	410	14580	FastTwin
21	icprimes	12250	4940	14140	370	14510	FastTwin
22	icprimes	15550	7450	22180	410	22590	FastTwin
23	icprimes	15550	7450	22190	350	22540	FastTwin
24	icprimes	15550	7450	22070	490	22560	FastTwin
25	icprimes	15550	7450	22140	440	22580	FastTwin
26	icprimes	15550	7450	22140	420	22560	FastTwin
27	icprimes	15550	7450	22150	420	22570	FastTwin
28	icprimes	15550	7450	22170	400	22570	FastTwin
29	icprimes	18340	9940	30260	370	30630	FastTwin
30	icprimes	18340	9940	30230	450	30680	FastTwin
31	icprimes	18340	9940	30230	420	30650	FastTwin
32	icprimes	18340	9940	30250	370	30620	FastTwin
33	icprimes	18340	9940	30240	370	30610	FastTwin
34	icprimes	18340	9940	30230	410	30640	FastTwin
35	icprimes	18340	9940	30210	420	30630	FastTwin
36	icprimes	49000	59960	191390	460	191850	FastTwin
37	icprimes	49000	59960	191460	570	192030	FastTwin
38	icprimes	49000	59960	191430	450	191880	FastTwin
39	icprimes	49000	59960	191490	410	191900	FastTwin
40	icprimes	49000	59960	191480	420	191900	FastTwin
41	icprimes	49000	59960	191560	310	191870	FastTwin
42	icprimes	49000	59960	191450	400	191850	FastTwin
43	icprimes	64150	99930	320230	440	320670	FastTwin
44	icprimes	64150	99930	320170	780	320950	FastTwin
45	icprimes	64150	99930	320670	640	321310	FastTwin
46	icprimes	64150	99930	320310	450	320760	FastTwin
47	icprimes	64150	99930	320290	390	320680	FastTwin
48	icprimes	64150	99930	320300	450	320750	FastTwin
49	icprimes	64150	99930	320330	340	320670	FastTwin

ICPRIMES FastTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	6.41874E+11	6.41874E+11	99999.99	0.0001
Error	47	3.92830E+05	8.35808E+03		
Corrected Total	48	6.41874E+11			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999999	0.108403	91.42256	84335.51

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	6.41874E+11	6.41874E+11	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	6.41874E+11	6.41874E+11	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-1433.067661	-87.81	0.0001	16.32059024
W	3.224729	8763.38	0.0001	0.00036798

ICPRIMES SlowTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	icprimes	3000	980	24680	450	25130	SlowTwin
2	icprimes	3000	980	24500	550	25050	SlowTwin
3	icprimes	3000	980	24860	230	25090	SlowTwin
4	icprimes	3000	980	24620	440	25060	SlowTwin
5	icprimes	3000	980	24670	390	25060	SlowTwin
6	icprimes	3000	980	24660	400	25060	SlowTwin
7	icprimes	3000	980	24640	390	25030	SlowTwin
8	icprimes	8800	2980	181840	440	182280	SlowTwin
9	icprimes	8800	2980	181360	410	181770	SlowTwin
10	icprimes	8800	2980	181740	480	182220	SlowTwin
11	icprimes	8800	2980	181360	440	181800	SlowTwin
12	icprimes	8800	2980	181480	300	181780	SlowTwin
13	icprimes	8800	2980	181410	370	181780	SlowTwin
14	icprimes	8800	2980	181340	410	181750	SlowTwin
15	icprimes	12250	4940	336700	440	337140	SlowTwin
16	icprimes	12250	4940	336910	430	337340	SlowTwin
17	icprimes	12250	4940	336900	350	337250	SlowTwin
18	icprimes	12250	4940	337550	690	338240	SlowTwin
19	icprimes	12250	4940	336750	410	337160	SlowTwin
20	icprimes	12250	4940	337070	460	337530	SlowTwin
21	icprimes	12250	4940	337730	370	338100	SlowTwin
22	icprimes	15550	7450	532580	410	532990	SlowTwin
23	icprimes	15550	7450	533770	370	534140	SlowTwin
24	icprimes	15550	7450	532550	410	532960	SlowTwin
25	icprimes	15550	7450	533720	440	534160	SlowTwin
26	icprimes	15550	7450	533710	390	534100	SlowTwin
27	icprimes	15550	7450	533910	330	534240	SlowTwin
28	icprimes	15550	7450	533220	390	533610	SlowTwin
29	icprimes	18340	9940	728670	470	729140	SlowTwin
30	icprimes	18340	9940	728830	420	729250	SlowTwin
31	icprimes	18340	9940	730400	390	730790	SlowTwin
32	icprimes	18340	9940	728920	370	729290	SlowTwin
33	icprimes	18340	9940	729590	540	730130	SlowTwin
34	icprimes	18340	9940	730410	460	730870	SlowTwin
35	icprimes	18340	9940	728930	320	729250	SlowTwin
36	icprimes	49000	59960	4664220	2990	4667210	SlowTwin
37	icprimes	49000	59960	4672050	1890	4673940	SlowTwin
38	icprimes	49000	59960	4666320	1970	4668290	SlowTwin
39	icprimes	49000	59960	4664010	2890	4666900	SlowTwin
40	icprimes	49000	59960	4663280	620	4663900	SlowTwin
41	icprimes	49000	59960	4669470	940	4670410	SlowTwin
42	icprimes	49000	59960	4663000	1240	4664240	SlowTwin
43	icprimes	64150	99930	7812920	3540	7816460	SlowTwin
44	icprimes	64150	99930	7812040	2270	7814310	SlowTwin
45	icprimes	64150	99930	7809160	2890	7812050	SlowTwin
46	icprimes	64150	99930	7812140	1420	7813560	SlowTwin
47	icprimes	64150	99930	7824890	770	7825660	SlowTwin
48	icprimes	64150	99930	7803210	760	7803970	SlowTwin
49	icprimes	64150	99930	7812170	6550	7818720	SlowTwin

ICPRIMES SlowTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	3.82563E+14	3.82563E+14	99999.99	0.0001
Error	47	3.54165E+08	7.53542E+06		
Corrected Total	48	3.82564E+14			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999999	0.134460	2745.072	2041554

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	3.82563E+14	3.82563E+14	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	3.82563E+14	3.82563E+14	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-52341.68654	-106.81	0.0001	490.0453866
W	78.72635	7125.21	0.0001	0.0110490

ICPRIMES CX
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	icprimes	3000	980	8440	320	8760	CX
2	icprimes	3000	980	8450	330	8780	CX
3	icprimes	3000	980	8420	360	8780	CX
4	icprimes	3000	980	8410	350	8760	CX
5	icprimes	3000	980	8450	290	8740	CX
6	icprimes	3000	980	8480	300	8780	CX
7	icprimes	3000	980	8460	340	8800	CX
8	icprimes	8800	2980	61930	470	62400	CX
9	icprimes	8800	2980	61950	350	62300	CX
10	icprimes	8800	2980	61880	410	62290	CX
11	icprimes	8800	2980	61960	320	62280	CX
12	icprimes	8800	2980	61900	390	62290	CX
13	icprimes	8800	2980	61900	380	62280	CX
14	icprimes	8800	2980	61940	280	62220	CX
15	icprimes	12250	4940	115390	540	115930	CX
16	icprimes	12250	4940	115450	310	115760	CX
17	icprimes	12250	4940	115460	270	115730	CX
18	icprimes	12250	4940	115450	340	115790	CX
19	icprimes	12250	4940	115410	360	115770	CX
20	icprimes	12250	4940	115470	410	115880	CX
21	icprimes	12250	4940	115430	400	115830	CX
22	icprimes	15550	7450	182670	390	183060	CX
23	icprimes	15550	7450	182670	410	183080	CX
24	icprimes	15550	7450	182630	340	182970	CX
25	icprimes	15550	7450	182630	410	183040	CX
26	icprimes	15550	7450	182660	400	183060	CX
27	icprimes	15550	7450	182700	330	183030	CX
28	icprimes	15550	7450	182770	450	183220	CX
29	icprimes	18340	9940	250060	390	250450	CX
30	icprimes	18340	9940	250130	400	250530	CX
31	icprimes	18340	9940	250190	330	250520	CX
32	icprimes	18340	9940	250140	380	250520	CX
33	icprimes	18340	9940	250070	340	250410	CX
34	icprimes	18340	9940	250070	400	250470	CX
35	icprimes	18340	9940	250090	350	250440	CX
36	icprimes	49000	59960	1601330	670	1602000	CX
37	icprimes	49000	59960	1601250	470	1601720	CX
38	icprimes	49000	59960	1601440	520	1601960	CX
39	icprimes	49000	59960	1601970	1940	1603910	CX
40	icprimes	49000	59960	1601560	390	1601950	CX
41	icprimes	49000	59960	1601730	380	1602110	CX
42	icprimes	49000	59960	1602070	400	1602470	CX
43	icprimes	64150	99930	2681440	3470	2684910	CX
44	icprimes	64150	99930	2681300	430	2681730	CX
45	icprimes	64150	99930	2681340	450	2681790	CX
46	icprimes	64150	99930	2681270	520	2681790	CX
47	icprimes	64150	99930	2681370	1280	2682650	CX
48	icprimes	64150	99930	2680910	600	2681510	CX
49	icprimes	64150	99930	2682160	450	2682610	CX

ICPRIMES CX
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	4.50744E+13	4.50744E+13	99999.99	0.0001
Error	47	1.40692E+07	2.99345E+05		
Corrected Total	48	4.50744E+13			

R-Square	C.V.	Root MSE	CPUTOT Mean
1.000000	0.078078	547.1241	700735.9

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	4.50744E+13	4.50744E+13	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	4.50744E+13	4.50744E+13	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-17998.08216	-184.27	0.0001	97.67160511
W	27.02298	12270.97	0.0001	0.00220219

ICPRIMES Mothra
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	icprimes	3000	980	79120	170	79290	Mothra
2	icprimes	3000	980	79050	180	79230	Mothra
3	icprimes	3000	980	79150	140	79290	Mothra
4	icprimes	3000	980	78940	210	79150	Mothra
5	icprimes	3000	980	79100	220	79320	Mothra
6	icprimes	3000	980	78980	190	79170	Mothra
7	icprimes	3000	980	78980	160	79140	Mothra
8	icprimes	8800	2980	583190	240	583430	Mothra
9	icprimes	8800	2980	587010	640	587650	Mothra
10	icprimes	8800	2980	587270	500	587770	Mothra
11	icprimes	8800	2980	587010	140	587150	Mothra
12	icprimes	8800	2980	586870	180	587050	Mothra
13	icprimes	8800	2980	587480	450	587930	Mothra
14	icprimes	8800	2980	587010	360	587370	Mothra
15	icprimes	12250	4940	1083570	180	1083750	Mothra
16	icprimes	12250	4940	1083580	240	1083820	Mothra
17	icprimes	12250	4940	1084800	1850	1086650	Mothra
18	icprimes	12250	4940	1083200	250	1083450	Mothra
19	icprimes	12250	4940	1083250	150	1083400	Mothra
20	icprimes	12250	4940	1083510	300	1083810	Mothra
21	icprimes	12250	4940	1083630	170	1083800	Mothra
22	icprimes	15550	7450	1711020	250	1711270	Mothra
23	icprimes	15550	7450	1712110	310	1712420	Mothra
24	icprimes	15550	7450	1712060	770	1712830	Mothra
25	icprimes	15550	7450	1727130	390	1727520	Mothra
26	icprimes	15550	7450	1712060	340	1712400	Mothra
27	icprimes	15550	7450	1711990	270	1712260	Mothra
28	icprimes	15550	7450	1711080	180	1711260	Mothra
29	icprimes	18340	9940	2341510	260	2341770	Mothra
30	icprimes	18340	9940	2380570	1220	2381790	Mothra
31	icprimes	18340	9940	2370420	2560	2372980	Mothra
32	icprimes	18340	9940	2354440	310	2354750	Mothra
33	icprimes	18340	9940	2365360	1650	2367010	Mothra
34	icprimes	18340	9940	2355870	1270	2357140	Mothra
35	icprimes	18340	9940	2355260	420	2355680	Mothra
36	icprimes	49000	59960	15142800	2080	15144880	Mothra
37	icprimes	49000	59960	15144020	3910	15147930	Mothra
38	icprimes	49000	59960	15059540	6350	15065890	Mothra
39	icprimes	49000	59960	15089560	30070	15119630	Mothra
40	icprimes	49000	59960	15065600	1560	15067160	Mothra
41	icprimes	49000	59960	15061090	4040	15065130	Mothra
42	icprimes	49000	59960	15023470	1020	15024490	Mothra
43	icprimes	64150	99930	25222970	5700	25228670	Mothra
44	icprimes	64150	99930	25225020	63120	25288140	Mothra
45	icprimes	64150	99930	25221740	8160	25229900	Mothra
46	icprimes	64150	99930	25272270	12760	25285030	Mothra
47	icprimes	64150	99930	25321750	45450	25367200	Mothra
48	icprimes	64150	99930	25213900	6050	25219950	Mothra
49	icprimes	64150	99930	25286750	3660	25290410	Mothra

ICPRIMES Mothra
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	4.00191E+15	4.00191E+15	99999.99	0.0001
Error	47	3.16115E+10	6.72584E+08		
Corrected Total	48	4.00194E+15			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999992	0.393031	25934.23	6598512

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	4.00191E+15	4.00191E+15	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	4.00191E+15	4.00191E+15	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-173801.2150	-37.54	0.0001	4629.732238
W	254.6256	2439.27	0.0001	0.104386

SUMSQRT FastTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	sumsqrt.	8500	980	1200	420	1620	FastTwin
2	sumsqrt.	8500	980	1170	380	1550	FastTwin
3	sumsqrt.	8500	980	1210	350	1560	FastTwin
4	sumsqrt.	8500	980	1200	370	1570	FastTwin
5	sumsqrt.	8500	980	1160	360	1520	FastTwin
6	sumsqrt.	8500	980	1110	470	1580	FastTwin
7	sumsqrt.	8500	980	1200	370	1570	FastTwin
8	sumsqrt.	58500	2940	6940	440	7380	FastTwin
9	sumsqrt.	58500	2940	6860	500	7360	FastTwin
10	sumsqrt.	58500	2940	7140	220	7360	FastTwin
11	sumsqrt.	58500	2940	7020	340	7360	FastTwin
12	sumsqrt.	58500	2940	6900	450	7350	FastTwin
13	sumsqrt.	58500	2940	6800	530	7330	FastTwin
14	sumsqrt.	58500	2940	7050	330	7380	FastTwin
15	sumsqrt.	106700	4950	12850	410	13260	FastTwin
16	sumsqrt.	106700	4950	12820	430	13250	FastTwin
17	sumsqrt.	106700	4950	12830	380	13210	FastTwin
18	sumsqrt.	106700	4950	12810	440	13250	FastTwin
19	sumsqrt.	106700	4950	12800	410	13210	FastTwin
20	sumsqrt.	106700	4950	12830	400	13230	FastTwin
21	sumsqrt.	106700	4950	12910	350	13260	FastTwin
22	sumsqrt.	164300	7490	20190	500	20690	FastTwin
23	sumsqrt.	164300	7490	20170	520	20690	FastTwin
24	sumsqrt.	164300	7490	20250	450	20700	FastTwin
25	sumsqrt.	164300	7490	20190	460	20650	FastTwin
26	sumsqrt.	164300	7490	20250	420	20670	FastTwin
27	sumsqrt.	164300	7490	20260	420	20680	FastTwin
28	sumsqrt.	164300	7490	20220	480	20700	FastTwin
29	sumsqrt.	221600	10000	27630	480	28110	FastTwin
30	sumsqrt.	221600	10000	27750	350	28100	FastTwin
31	sumsqrt.	221600	10000	27610	500	28110	FastTwin
32	sumsqrt.	221600	10000	27770	360	28130	FastTwin
33	sumsqrt.	221600	10000	27750	340	28090	FastTwin
34	sumsqrt.	221600	10000	27640	470	28110	FastTwin
35	sumsqrt.	221600	10000	27600	490	28090	FastTwin
36	sumsqrt.	1283000	60020	174690	350	175040	FastTwin
37	sumsqrt.	1283000	60020	174700	320	175020	FastTwin
38	sumsqrt.	1283000	60020	174280	830	175110	FastTwin
39	sumsqrt.	1283000	60020	174380	2270	176650	FastTwin
40	sumsqrt.	1283000	60020	173850	390	174240	FastTwin
41	sumsqrt.	1283000	60020	173860	530	174390	FastTwin
42	sumsqrt.	1283000	60020	173730	510	174240	FastTwin
43	sumsqrt.	2070000	100020	291790	430	292220	FastTwin
44	sumsqrt.	2070000	100020	291660	530	292190	FastTwin
45	sumsqrt.	2070000	100020	291870	320	292190	FastTwin
46	sumsqrt.	2070000	100020	291780	360	292140	FastTwin
47	sumsqrt.	2070000	100020	291880	290	292170	FastTwin
48	sumsqrt.	2070000	100020	291790	460	292250	FastTwin
49	sumsqrt.	2070000	100020	291860	320	292180	FastTwin

SUMSQRT FastTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	5.32491E+11	5.32491E+11	99999.99	0.0001
Error	47	4.37574E+06	9.31009E+04		
Corrected Total	48	5.32496E+11			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999992	0.396927	305.1244	76871.63

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	5.32491E+11	5.32491E+11	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	5.32491E+11	5.32491E+11	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-1273.143045	-23.37	0.0001	54.47658069
W	2.934621	2391.55	0.0001	0.00122708

SUMSQRT SlowTwin
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	sumsqrt.	8500	980	14240	510	14750	SlowTwin
2	sumsqrt.	8500	980	14400	420	14820	SlowTwin
3	sumsqrt.	8500	980	14310	450	14760	SlowTwin
4	sumsqrt.	8500	980	14440	350	14790	SlowTwin
5	sumsqrt.	8500	980	14350	410	14760	SlowTwin
6	sumsqrt.	8500	980	14300	460	14760	SlowTwin
7	sumsqrt.	8500	980	14320	450	14770	SlowTwin
8	sumsqrt.	58500	2940	111880	440	112320	SlowTwin
9	sumsqrt.	58500	2940	111880	390	112270	SlowTwin
10	sumsqrt.	58500	2940	111960	320	112280	SlowTwin
11	sumsqrt.	58500	2940	111840	420	112260	SlowTwin
12	sumsqrt.	58500	2940	111830	550	112380	SlowTwin
13	sumsqrt.	58500	2940	111830	410	112240	SlowTwin
14	sumsqrt.	58500	2940	112030	200	112230	SlowTwin
15	sumsqrt.	106700	4950	210930	430	211360	SlowTwin
16	sumsqrt.	106700	4950	210910	500	211410	SlowTwin
17	sumsqrt.	106700	4950	210880	430	211310	SlowTwin
18	sumsqrt.	106700	4950	210790	520	211310	SlowTwin
19	sumsqrt.	106700	4950	210870	580	211450	SlowTwin
20	sumsqrt.	106700	4950	210970	340	211310	SlowTwin
21	sumsqrt.	106700	4950	210920	440	211360	SlowTwin
22	sumsqrt.	164300	7490	335560	570	336130	SlowTwin
23	sumsqrt.	164300	7490	335740	540	336280	SlowTwin
24	sumsqrt.	164300	7490	335700	550	336250	SlowTwin
25	sumsqrt.	164300	7490	335510	490	336000	SlowTwin
26	sumsqrt.	164300	7490	335860	410	336270	SlowTwin
27	sumsqrt.	164300	7490	335530	570	336100	SlowTwin
28	sumsqrt.	164300	7490	335950	610	336560	SlowTwin
29	sumsqrt.	221600	10000	460100	690	460790	SlowTwin
30	sumsqrt.	221600	10000	460130	460	460590	SlowTwin
31	sumsqrt.	221600	10000	460420	420	460840	SlowTwin
32	sumsqrt.	221600	10000	460160	540	460700	SlowTwin
33	sumsqrt.	221600	10000	459970	650	460620	SlowTwin
34	sumsqrt.	221600	10000	460450	490	460940	SlowTwin
35	sumsqrt.	221600	10000	460160	420	460580	SlowTwin
36	sumsqrt.	1283000	60020	2925490	910	2926400	SlowTwin
37	sumsqrt.	1283000	60020	2927760	940	2928700	SlowTwin
38	sumsqrt.	1283000	60020	2927110	800	2927910	SlowTwin
39	sumsqrt.	1283000	60020	2928180	810	2928990	SlowTwin
40	sumsqrt.	1283000	60020	2924420	1030	2925450	SlowTwin
41	sumsqrt.	1283000	60020	2927930	1080	2929010	SlowTwin
42	sumsqrt.	1283000	60020	2927590	810	2928400	SlowTwin
43	sumsqrt.	2070000	100020	4894770	1410	4896180	SlowTwin
44	sumsqrt.	2070000	100020	4909110	2230	4911340	SlowTwin
45	sumsqrt.	2070000	100020	4894140	1560	4895700	SlowTwin
46	sumsqrt.	2070000	100020	4894320	1300	4895620	SlowTwin
47	sumsqrt.	2070000	100020	4895000	1830	4896830	SlowTwin
48	sumsqrt.	2070000	100020	4895370	2510	4897880	SlowTwin
49	sumsqrt.	2070000	100020	4900720	9960	4910680	SlowTwin

SUMSQRT SlowTwin
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	1.50447E+14	1.50447E+14	99999.99	0.0001
Error	47	3.23941E+08	6.89236E+06		
Corrected Total	48	1.50447E+14			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999998	0.205017	2625.330	1280544

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	1.50447E+14	1.50447E+14	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	1.50447E+14	1.50447E+14	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-32972.37616	-70.35	0.0001	468.7236541
W	49.32732	4672.05	0.0001	0.0105580

SUMSQRT Mothra
Input Data (Sorted by W)

OBS	P	N	W	U	S	CPUTOT	VNAME
1	sumsqrt.	8500	980	54780	160	54940	Mothra
2	sumsqrt.	8500	980	54840	230	55070	Mothra
3	sumsqrt.	8500	980	54770	180	54950	Mothra
4	sumsqrt.	8500	980	54780	220	55000	Mothra
5	sumsqrt.	8500	980	54830	110	54940	Mothra
6	sumsqrt.	8500	980	54800	140	54940	Mothra
7	sumsqrt.	8500	980	54810	160	54970	Mothra
8	sumsqrt.	58500	2940	436340	250	436590	Mothra
9	sumsqrt.	58500	2940	436120	180	436300	Mothra
10	sumsqrt.	58500	2940	436390	270	436660	Mothra
11	sumsqrt.	58500	2940	436240	300	436540	Mothra
12	sumsqrt.	58500	2940	436040	260	436300	Mothra
13	sumsqrt.	58500	2940	436100	180	436280	Mothra
14	sumsqrt.	58500	2940	436110	270	436380	Mothra
15	sumsqrt.	106700	4950	822950	290	823240	Mothra
16	sumsqrt.	106700	4950	822540	270	822810	Mothra
17	sumsqrt.	106700	4950	822800	450	823250	Mothra
18	sumsqrt.	106700	4950	823590	1220	824810	Mothra
19	sumsqrt.	106700	4950	822340	300	822640	Mothra
20	sumsqrt.	106700	4950	822760	430	823190	Mothra
21	sumsqrt.	106700	4950	822340	260	822600	Mothra
22	sumsqrt.	164300	7490	1312110	360	1312470	Mothra
23	sumsqrt.	164300	7490	1312210	400	1312610	Mothra
24	sumsqrt.	164300	7490	1312410	470	1312880	Mothra
25	sumsqrt.	164300	7490	1311980	240	1312220	Mothra
26	sumsqrt.	164300	7490	1312540	330	1312870	Mothra
27	sumsqrt.	164300	7490	1312600	700	1313300	Mothra
28	sumsqrt.	164300	7490	1311990	440	1312430	Mothra
29	sumsqrt.	221600	10000	1799890	550	1800440	Mothra
30	sumsqrt.	221600	10000	1800810	1460	1802270	Mothra
31	sumsqrt.	221600	10000	1816820	620	1817440	Mothra
32	sumsqrt.	221600	10000	1799540	500	1800040	Mothra
33	sumsqrt.	221600	10000	1799620	660	1800280	Mothra
34	sumsqrt.	221600	10000	1799350	440	1799790	Mothra
35	sumsqrt.	221600	10000	1799530	650	1800180	Mothra
36	sumsqrt.	1283000	60020	11438980	1990	11440970	Mothra
37	sumsqrt.	1283000	60020	11460500	3130	11463630	Mothra
38	sumsqrt.	1283000	60020	11445200	2090	11447290	Mothra
39	sumsqrt.	1283000	60020	11452570	2320	11454890	Mothra
40	sumsqrt.	1283000	60020	11541300	30540	11571840	Mothra
41	sumsqrt.	1283000	60020	11464030	20900	11484930	Mothra
42	sumsqrt.	1283000	60020	11438530	2960	11441490	Mothra
43	sumsqrt.	2070000	100020	19058660	3500	19062160	Mothra
44	sumsqrt.	2070000	100020	19164580	19740	19184320	Mothra
45	sumsqrt.	2070000	100020	19155030	4930	19159960	Mothra
46	sumsqrt.	2070000	100020	19156020	3510	19159530	Mothra
47	sumsqrt.	2070000	100020	22094060	45470	.	Mothra
48	sumsqrt.	2070000	100020	19133820	6170	19139990	Mothra
49	sumsqrt.	2070000	100020	19150690	3820	19154510	Mothra

SUMSQRT Mothra
Linear Regression Model: CPUTOT = W

General Linear Models Procedure

Number of observations in data set = 49
NOTE: Due to missing values, only 48 observations can be used in this analysis.

Dependent Variable: CPUTOT

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	2.09525E+15	2.09525E+15	99999.99	0.0001
Error	46	2.90112E+10	6.30678E+08		
Corrected Total	47	2.09528E+15			

R-Square	C.V.	Root MSE	CPUTOT Mean
0.999986	0.532962	25113.30	4712024

Source	DF	Type I SS	Mean Square	F Value	Pr > F
W	1	2.09525E+15	2.09525E+15	99999.99	0.0001

Source	DF	Type III SS	Mean Square	F Value	Pr > F
W	1	2.09525E+15	2.09525E+15	99999.99	0.0001

Parameter	Estimate	T for H0: Parameter=0	Pr > |T|	Std Error of Estimate
INTERCEPT	-128625.8801	-28.62	0.0001	4493.574506
W	192.8578	1822.69	0.0001	0.105809

Appendix H

Conservative Low Estimation Equations

This appendix contains the estimation equations used in determining the "conservative low" `IDEAL` baseline values, denoted as T below, used in the aggregate performance study described in Chapter IV.

`CHI` Estimation Equation

$$T = \beta_0 \ + \ \beta_1 S_t,$$

where T is the conservative low baseline time and S_t is the total statement tally (or $\sum_{i=1}^{20} S_i$, where S_i is the statement tally for statement i). Determined through least squares estimation ($R^2 = 1.00$): $\beta_0 = 6.551625373$ ($\rho = .0001$) and $\beta_1 = 0.001375889$ ($\rho = .0001$).

These values were used as-is without further adjustment.

`CPRIMES` Estimation Equation

$$T = \begin{cases} \beta_0 \ + \ \beta_1 S_t & \text{, if greater than zero} \\ 3.0 & \text{, otherwise} \end{cases}$$

where T is the conservative low baseline time and S_t is the total statement tally (or $\sum_{i=1}^{18} S_i$, where S_i is the statement tally for statement i). Determined through least squares estimation ($R^2 = 1.00$): $\beta_0 = -16.02514297$ ($\rho = .1253$) and $\beta_1 = 0.00082288$ ($\rho = .0001$).

ICHI Estimation Equation

$$T = \beta_0 \ + \ \beta_1 S_t,$$

where T is the conservative low baseline time and S_t is the total statement tally (or $\sum_{i=1}^{20} S_i$, where S_i is the statement tally for statement i). Determined through least squares estimation ($R^2 = 1.00$): $\beta_0 = 7.023234280$ ($\rho = .0001$) and $\beta_1 = 0.001351291$ ($\rho = .0001$).

These values were used as-is without further adjustment.

ICPRIMES Estimation Equation

$$I = \beta_0 \ + \ \beta_1 S_t \ + \ \beta_2 S_5 \ + \ \beta_3 S_{14},$$

where I is the *initial* conservative low estimate, S_t is the total statement tally (or $\sum_{i=1}^{17} S_i$), and where S_i is the statement tally for statement i. Determined through least squares estimation ($R^2 = 1.00$): $\beta_0 = 8.119724483$ ($\rho = .0696$), $\beta_1 = 0.000508531$ ($\rho = .0001$), $\beta_2 = -0.021067316$ ($\rho = .0642$), and $\beta_3 = 0.074357208$ ($\rho = .1033$).

$$T = \begin{cases} 15.0 & \text{, if } -36.0 \le I < 0 \\ 25.0 & \text{, if } -54.0 \le I < -36.0 \\ 30.0 & \text{, if } -67.0 \le I < -54.0 \\ 23256.0 & \text{, if } I < -67.0 \text{ or } I > 23256 \\ I & \text{, otherwise} \end{cases}$$

where T is the conservative low baseline time.

<u>SUMSQRT Estimation Equation</u>

$$I = \beta_0 \ + \ \beta_1 S_4 \ + \ \beta_2 S_9 \ + \ \beta_3 S_{15},$$

where I is the *initial* conservative low estimate and where S_i is the statement tally for statement i. Determined through least squares estimation ($R^2 = 1.00$): $\beta_0 = 9.738606734$ ($\rho = .0131$), $\beta_1 = -0.002172102$ ($\rho = .0001$), $\beta_2 = 0.003803638$ ($\rho = .0001$), and $\beta_3 = -0.000287731$ ($\rho = .0005$).

$$T = \begin{cases} 22579.3 & \text{, if } I \approx -101793.8 \\ 105.2 & \text{, if } I \approx -117.3 \\ I & \text{, otherwise} \end{cases}$$

where T is the conservative low baseline time.

Appendix I

The TUMS C Language Definition

The grammar that defines the C language processed by TUMS is given below. The grammar is expressed in the factored extended BNF formalism used by the tool Ast [55]. Following the grammar are additional comments.

```
PARSER

GLOBAL {
/* From GLOBAL section in c.grammar */

#include "SS.h"
}


BEGIN {
/* From BEGIN section in c.grammar */
}


PREC
   LEFT '||'
   LEFT '&&'
   LEFT '|'
   LEFT '^'
   LEFT '&'
   LEFT '==' '!='
   LEFT '<' '>' '<=' '>='
   LEFT '<<' '>>'
   LEFT '+' '-'
   LEFT '*' '/' '%'
   NONE if
   NONE else

PROPERTY INPUT


RULE

/* === Translation unit (highest level) section ======================== */

START = translation_unit .

translation_unit = <
   TopLevelDecl  = top_level_declaration .
   TopLevelDecls = translation_unit top_level_declaration .
> .


/* === Declarations section ============================================= */

top_level_declaration = <
   Declaration          = declaration .
   Function_definition  = function_definition .
> .

function_definition  =
   function_specifier SS_funcb '{' Function_body '}' .

SS_funcb = .

function_specifier = <
   Hdr0 =                         declarator REC_off                    REC_on .
   Hdr1 =                         declarator REC_off Declaration_list REC_on .
   Hdr2 = Declaration_specifiers declarator REC_off                    REC_on .
   Hdr3 = Declaration_specifiers declarator REC_off Declaration_list REC_on .
> .

REC_off = .
REC_on  = .
```

```
Function_body = <
   NoBody        = .
   StatsOnlyBody = statement_list .
   VarOnlyBody   = Declaration_list .
   VarStatsBody  = Declaration_list statement_list .
> .

Declaration_list = declaration_list .

declaration_list = <
   Decl  = declaration .
   Decls = declaration_list declaration .
> .

declaration = Declaration_specifiers Declarator_list ';' .

Declaration_specifiers = declaration_specifiers .

declaration_specifiers = <
   storage_class_specifier = <
      Auto     = auto .
      Register = register .
      Static   = static .
      Extern   = extern .
      Typedef  = typedef .
   > .
   Storage_class_specified = storage_class_specifier declaration_specifiers .
   type_specifier = <
      Root_type_specifier = <
         Void                       = void .
         Char                       = char .
         Int                        = int .
         Float                      = float .
         Double                     = double .
         Struct_or_union_specifier = struct_or_union_specifier .
         Typedef_name               = TYPEDEF_NAME .
      > .
      Type_adjective = <
         Short    = short .
         Long     = long .
         Signed   = signed .
         Unsigned = unsigned .
      > .
   > .
   Type_specified = type_specifier declaration_specifiers .
> .

struct_or_union_specifier = <
   = struct_or_union '{' struct_declaration_list '}' .
   = struct_or_union tag '{' struct_declaration_list '}' .
   = struct_or_union tag .
> .

tag = In_scope_identifier .

struct_or_union = <
   = struct .
   = union .
> .

struct_declaration_list = <
   struct_declaration  = specifier_qualifier_list struct_declarator_list ';' .
   = struct_declaration_list struct_declaration .
> .

specifier_qualifier_list = <
   = type_specifier .
   = type_specifier specifier_qualifier_list .
> .

struct_declarator_list = <
   struct_declarator  = declarator .
   Struct_declarators = struct_declarator_list ',' struct_declarator .
> .
```

```
declarator = <
   direct_declarator = <
      simple_declarator = In_scope_identifier .
      Paren_declarator  = '(' declarator ')' .
      array_declarator = <
         Empty_array_declarator = direct_declarator '[' ']' .
         Array_declarator       = direct_declarator '[' INTEGER_CONSTANT ']' .
      > .
      Function_declarator = function_declarator .
   > .
   Pointer_declarator = pointer direct_declarator .
> .

function_declarator = <
   Newstyle_func = direct_declarator SS_parms '(' parameter_type_list ')' .
   Nolist_func   = direct_declarator SS_parms '('                     ')' .
   Oldstyle_func = direct_declarator SS_parms '(' Identifier_list     ')' .
> .

SS_parms = .

Declarator_list = <
   Declarator  = declarator .
   Declarators = Declarator_list ',' declarator .
> .

pointer = <
   Star  = '*' .
   Stars = '*' pointer .
> .

parameter_type_list = parameter_list .

parameter_declaration = <
   Formal_parameter        = Declaration_specifiers declarator .
   Parameter_type          = Declaration_specifiers .
   Parameter_abstract_type = Declaration_specifiers abstract_declarator .
> .

parameter_list = <
   Parameter  = parameter_declaration .
   Parameters = parameter_list ',' parameter_declaration .
> .

Identifier_list = identifier_list .

identifier_list = <
   Formal_identifier  = In_scope_identifier .
   Formal_identifiers = identifier_list ',' In_scope_identifier .
> .

type_name = <
   = specifier_qualifier_list .
   = specifier_qualifier_list abstract_declarator .
> .

abstract_declarator = <
   = pointer .
   direct_abstract_declarator = <
      = '(' abstract_declarator ')' .
      = '[' ']' .
      = '[' constant_expr ']' .
      = direct_abstract_declarator '[' ']' .
      = direct_abstract_declarator '[' constant_expr ']' .
      = '(' ')' .
      = '(' parameter_type_list ')' .
      = direct_abstract_declarator '(' ')' .
      = direct_abstract_declarator '(' parameter_type_list ')' .
   > .
   = pointer direct_abstract_declarator .
> .
```

```
/* === Executable section =============================================== */

statement_list = <
   Stat  = statement .
   Stats = statement_list statement .
> .

label = <
   Named_label   = In_scope_identifier ':' .
   Case_label    = case constant_expr ':' .
   Default_label = default ':' .
> .

Label_list = <
   Label  = label .
   Labels = Label_list label .
> .

statement = <
   labeled_statement    = Label_list Stmt .
   Unlabeled_statement  =            Stmt .
> .

Stmt = <
   expr_statement = <
      Null_stat = ';' .
      Expr_stat = expr ';' .
   > .
   compound_statement = <
      Empty     = '{' '}' .
      StatsOnly = '{' statement_list '}' .
   > .
   selection_statement = <
      IfThen     = if '(' expr ')' Then:statement                     PREC if .
      IfThenElse = if '(' expr ')' Then:statement else Else:statement PREC else.
      Switch     = switch '(' expr ')' statement .
   > .
   iteration_statement = <
      While   = while '(' expr ')' statement .
      DoWhile = do statement while '(' expr ')' ';' .
      For000  = for '('           ';'           ';'           ')' statement .
      For001  = for '('           ';'           ';' Post:expr ')' statement .
      For010  = for '('           ';' Pred:expr ';'           ')' statement .
      For011  = for '('           ';' Pred:expr ';' Post:expr ')' statement .
      For100  = for '(' Init:expr ';'           ';'           ')' statement .
      For101  = for '(' Init:expr ';'           ';' Post:expr ')' statement .
      For110  = for '(' Init:expr ';' Pred:expr ';'           ')' statement .
      For111  = for '(' Init:expr ';' Pred:expr ';' Post:expr ')' statement .
   > .
   jump_statement = <
      Goto       = goto identifier ';' .
      Continue   = continue ';' .
      Break      = break ';' .
      Return     = return ';' .
      ReturnExpr = return expr ';' .
   > .
> .

constant_expr  = conditional_expr .

expr = <
   assignment_expr = <
      conditional_expr = <
         binary_expr = <
            Logical_or     = Left:binary_expr '||' Right:binary_expr .
            Logical_and    = Left:binary_expr '&&' Right:binary_expr .
            Inclusive_or   = Left:binary_expr '|'  Right:binary_expr .
            Exclusive_or   = Left:binary_expr '^'  Right:binary_expr .
            And            = Left:binary_expr '&'  Right:binary_expr .
            Equal          = Left:binary_expr '==' Right:binary_expr .
            Not_equal      = Left:binary_expr '!=' Right:binary_expr .
            Lt             = Left:binary_expr '<'  Right:binary_expr .
            Gt             = Left:binary_expr '>'  Right:binary_expr .
            Le             = Left:binary_expr '<=' Right:binary_expr .
```

```
            Ge             = Left:binary_expr '>=' Right:binary_expr .
            Shift_left     = Left:binary_expr '<<' Right:binary_expr .
            Shift_right    = Left:binary_expr '>>' Right:binary_expr .
            Add            = Left:binary_expr '+'  Right:binary_expr .
            Subtract       = Left:binary_expr '-'  Right:binary_expr .
            Multiply       = Left:binary_expr '*'  Right:binary_expr .
            Divide         = Left:binary_expr '/'  Right:binary_expr .
            Modulus        = Left:binary_expr '%'  Right:binary_expr .
            cast_expr = <
               unary_expr = <
                  postfix_expr = <
                     primary_expr = <
                        = identifier .
                        = constant .
                        = string_literal .
                        = '(' expr ')' .
                     > .
                     Subscript       = postfix_expr '[' expr ']' .
                     CallNoArgs      = postfix_expr '(' ')' .
                     Call            = postfix_expr '(' argument_expr_list ')' .
                     Direct_select   = postfix_expr '.' identifier .
                     Pointer_select  = postfix_expr '->' identifier .
                     Postfix_inc     = postfix_expr '++' .
                     Postfix_dec     = postfix_expr '--' .
                  > .
                  Prefix_inc  = '++' unary_expr .
                  Prefix_dec  = '--' unary_expr .
                  Address     = '&'  cast_expr .
                  Indirection = '*'  cast_expr .
                  Unary_plus  = '+'  cast_expr .
                  Unary_minus = '-'  cast_expr .
                  Complement  = '~'  cast_expr .
                  Not         = '!'  cast_expr .
                  Sizeof_expr = sizeof unary_expr .
                  Sizeof_type = sizeof '(' type_name ')' .
               > .
               Cast = '(' type_name ')' cast_expr .
            > .
         > .
      Conditional = binary_expr '?' expr ':' conditional_expr .
      > .
      Asgn              = unary_expr '='   assignment_expr .
      Inclusive_or_asgn = unary_expr '|='  assignment_expr .
      Exclusive_or_asgn = unary_expr '^='  assignment_expr .
      And_asgn          = unary_expr '&='  assignment_expr .
      Shift_left_asgn   = unary_expr '<<=' assignment_expr .
      Shift_right_asgn  = unary_expr '>>=' assignment_expr .
      Add_asgn          = unary_expr '+='  assignment_expr .
      Subtract_asgn     = unary_expr '-='  assignment_expr .
      Multiply_asgn     = unary_expr '*='  assignment_expr .
      Divide_asgn       = unary_expr '/='  assignment_expr .
      Modulus_asgn      = unary_expr '%='  assignment_expr .
   > .
   Comma_expr = expr ',' assignment_expr .
> .

argument_expr_list = <
   Arg  = assignment_expr .
   Args = argument_expr_list ',' assignment_expr .
> .

identifier = IDENTIFIER .

In_scope_identifier = IDENTIFIER .

string_literal = STRING .

constant = <
   float_constant = FLOATING_CONSTANT .
   int_constant   = INTEGER_CONSTANT .
   char_constant  = CHARACTER_CONSTANT .
> .
```

```
Terminal :  <
   IDENTIFIER          : 1  [Object : tObject] { Object := NoObject; } .
   TYPEDEF_NAME        : 2  [Object : tObject] { Object := NoObject; } .
   FLOATING_CONSTANT   : 4  [Object : tObject] [Value : tLDBL]
                            { Object := NoObject;  Value := 0.0; } .
   INTEGER_CONSTANT    : 5  [Object : tObject] [Value : tULONG]
                            { Object := NoObject;  Value := 0; } .
   CHARACTER_CONSTANT  : 6  [Object : tObject] [Value : string]
                            { Object := NoObject;  Value := '\0'; } .
   STRING              : 7  [Object : tObject] [Value : string]
                            { Object := NoObject;  Value := '\0'; } .
> .

/* === End c.grammar ===================================================== */
```

- *Lexical:* The following rare lexical constructs are not included: trigraphs, wide character constants, and multibyte characters.
- *Syntax:* Removed useless untyped declarations. For example, the declaration "x;" is not valid; a type must be specified in all declarations.
- *Syntax:* Removed declaration initializers.
- *Syntax:* Removed declarations local to compound statements.
- *Syntax:* Eliminated `enum` processing.
- *Syntax:* Removed the type qualifiers `const` and `volatile`.
- *Syntax:* Disallowed the use of the ellipsis (...) in parameter type lists.
- *Syntax:* Removed support for bitsets, structures, and unions.
- *Syntax:* Restricted array extents to be integer constants.
- *Semantic:* Disallowed the use of the same `typedef` name in more than one context. Also, `typedef` names are restricted to use outside the target function.
- *Semantic:* Used only a single symbol space for all identifiers. In particular, labels and `struct` tags are stored in the same name space as variables.
- *Semantic:* Removed non-local jumps.
- *Semantic:* Do not support user-defined exception handling.

LITERATURE CITED

1. Glenford J. Myers. *The Art of Software Testing*. John Wiley and Sons, New York, NY, 1979.

2. Elaine J. Weyuker. In Defense of Coverage Criteria. In *Proceedings of the Eleventh International Conference on Software Engineering*, page 361, Pittsburgh, PA, May 16–18 1989. IEEE Computer Society Press.

3. J. W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

4. Elaine J. Weyuker. The Complexity of Data Flow Criteria for Test Data Selection. *Information Processing Letters*, 19(2):103–109, August 1984.

5. Simeon C. Ntafos. On Required Element Testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

6. Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

7. Phyllis G. Frankl and Elaine J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, October 1988.

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

9. 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.

10. Allen Troy Acree, Jr. *On Mutation*. Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, August 1980.

11. Timothy Alan Budd. *Mutation Analysis of Program Test Data*. Ph.D. dissertation, Yale University, New Haven, CT, May 1980.

12. Richard A. DeMillo. Test Adequacy and Program Mutation. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 355–356, Pittsburgh, PA, May 16–18 1989. IEEE Computer Society Press.

13. Richard A. DeMillo and A. Jefferson Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, SE-17(9):900–910, September 1991.

14. Timothy A. Budd. Mutation Analysis: Ideas, Examples, Problems and Prospects. In B. Chandrasekaran and S. Radicchi, editors, *Proceedings of the Summer School on Computer Program Testing*, Computer Program Testing, pages 129–148, SOGESTA, Urbino, Italy, June 29–July 3 1981. North-Holland Publishing Company, Amsterdam, Netherlands.

15. James E. Burns. Stability of Test Data from Program Mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 324–332, Ft. Lauderdale, FL, December 18–20 1978. IEEE Computer Society Press.

16. Moheb Ramzy Girgis and Martin R. Woodward. An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria. In *Proceedings of the Workshop on Software Testing*, pages 64–73, Banff, Alberta, Canada, July 1986. IEEE Computer Society Press.

17. Larry J. Morell. Theoretical Insights into Fault-Based Testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 45–62, Banff, Alberta, Canada, July 19–21 1988. IEEE Computer Society Press.

18. Larry J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, SE-16(8):844–857, August 1990.

19. Richard A. DeMillo and Aditya P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software. Technical Report SERC-TR-92-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, February 9 1991.

20. Weichen Eric Wong. *On Mutation and Data Flow.* Ph.D. dissertation, Purdue University, West Lafayette, IN, December 1993.

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

22. A. Jefferson Offutt and Scott V. Fichter. A Parallel Interpreter for the Mothra Mutation Testing System. Technical Report 92-100, Computer Science Department, Clemson University, Clemson, SC, January 1992.

23. 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.

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

25. 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.

26. Akihiko Tanaka. *Equivalence Testing for FORTRAN Mutation System Using Data Flow Analysis.* Master's thesis, Georgia Institute of Technology, Atlanta, GA, December 1981. (*also* Technical Report GIT-ICS-82/10, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA).

27. A. Jefferson Offutt and William Michael Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *The Journal of Software Testing, Verification, and Reliability*, 4(3):131–154, September 1994.

28. A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An Experimental Determination of Sufficient Mutation Operators. Technical Report ISSE-TR-94-100, Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA, January 1994.

29. ByoungJu Choi and Aditya P. Mathur. Use of Fifth Generation Computers for High Performance Reliable Software Testing (Final Report). Technical Report SERC-TR-72-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, April 10 1990.

30. 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.

31. A. T. Acree. CPMS User's Guide. Technical report GIT-ICS-79/04, Georgia Institute of Technology, Atlanta, GA, April 1979.

32. D. M. St. Andre. Pilot Mutation System (PIMS) User's Manual. Technical report GIT-ICS-79/04, Georgia Institute of Technology, Atlanta, GA, April 1979.

33. 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.

34. 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.

35. 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.

36. Mehmet Ṣahinoğlu and Eugene H. Spafford. A Bayes Sequential Statistical Procedure for Approving Software Products. In Wolfgang Ehrenberger, editor, *Proceedings of the IFIP Conference on Approving Software Products (ASP–90)*, pages 43–56, Garmisch-Partenkirchen, Germany, September 1990. Elsevier/North Holland, New York.

37. William Hsu, Mehmet Ṣahinoğlu, and Eugene H. Spafford. An Experimental Approach to Statistical Mutation-Based Testing. Technical Report SERC-TR-63-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, April 10 1990.

38. Aditya P. Mathur. Performance, Effectiveness, and Reliability Issues in Software Testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference (COMPSAC)*, pages 604–605, Tokyo, Japan, September 11–13 1991. IEEE Computer Society Press.

39. A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An Experimental Evaluation of Selective Mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, Baltimore, MD, May 17–21 1993. IEEE Computer Society Press.

40. 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.

41. 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.

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

43. Christian N. Zapf. MedusaMothra – A Distributed Interpreter for the Mothra Mutation Testing System. M.S. thesis, Clemson University, Clemson, SC, August 1993.

44. Stewart N. Weiss and Vladimir N. Fleyshgakker. Improved Serial Algorithms for Mutation Analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 93)*, pages 149–158, Cambridge, MA, June 28–30 1993. ACM SIGSOFT, ACM Press.

45. Vladimir N. Fleyshgakker and Stewart N. Weiss. Efficient Mutation Analysis: A New Approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 94)*, pages 185–195, Seattle, WA, August 17–19 1994. ACM SIGSOFT, ACM Press.

46. 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.

47. Edward William Krauser, Jr. *Compiler-Integrated Software Testing*. Ph.D. dissertation, Purdue University, West Lafayette, IN, December 1991. (*also* Technical Report SERC-TR-118-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

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

49. 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.

50. Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation Analysis Using Mutant Schemata. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 93)*, pages 139–148, Cambridge, MA, June 28–30 1993. ACM SIGSOFT, ACM Press.

51. 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.

52. 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.

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

54. Josef Grosch and Helmut Emmelmann. *A Tool Box for Compiler Construction*, volume 477 of *Lecture Notes in Computer Science*, pages 106–116. Springer-Verlag, New York, October 1990.

55. Josef Grosch. Ast—A Generator for Abstract Syntax Trees. Compiler Generation Report 15, GMD Forschungstelle an der Universität Karlsrhuhe, Karlsruhe, Germany, August 3 1992.

56. Josef Grosch. Efficient Generation of Lexical Analyzers. *Software—Practice and Experience*, 19(11):1089–1103, November 1989.

57. Josef Grosch. Lalr—a Generator for Efficient Parsers. *Software—Practice and Experience*, 20(11):1115–1135, November 1990.

58. Josef Grosch. Puma—A Generator for the Transformation of Attributed Trees. Compiler Generation Report 26, GMD Forschungstelle an der Universität Karlsrhuhe, Karlsruhe, Germany, November 22 1991.

59. C. A. R. Hoare. Algorithm 65: FIND. *Communications of the ACM*, 4(1):321–322, July 1961.

60. C. A. R. Hoare. Proof of a Program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

61. Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

62. Peter Collinson. Tools for Program Development. *EXE*, 6(5):89–92, October 1991.

63. Lyman Ott. *An Introduction to Statistical Methods and Data Analysis*. PWS-Kent Publishing Company, Boston, MA, third edition, 1988.

64. Rudolf J. Freund and Ramon C. Littell. *SAS for Linear Models: a guide to the ANOVA and GLM procedures*. SAS Institute Inc., Cary, NC, 1981.

65. Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill, New York, NY, 1985.

66. William E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

67. A. Jefferson Offutt and Stephen D. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, SE-20(5):337–344, May 1994.