DESIGN OF MUTANT OPERATORS FOR THE C PROGRAMMING LANGUAGE[†]

Hiralal Agrawal
Richard A. DeMillo
Bob Hathaway
William Hsu
Wynne Hsu
E.W. Krauser
R. J. Martin
Aditya P. Mathur
Eugene Spafford

Software Engineering Research Center
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907

Revision 1.02

March 20, 1989

[†]This work was supported by Rome Air Development Center Contract No. F 30602-85-C-0255 and Georgia Institute of Technology subcontract B-10-653-1.

"Mutation decides the course of humanity."

—— *Not a software engineer*

Contents

List of Figures

List of Tables

Abstract

Mutation analysis is a method for reliable testing of large software systems. It provides a method for assessing the adequacy of test data. *Mothra* is a mutation analysis based software testing environment that currently supports the testing of Fortran 77 programs. Work is underway to enhance this tool along several dimensions. One of these is the addition of *multilingual* capability. C is one of the languages that we plan to support.

This report describes the mutant operators designed for the proposed ANSI C programming language. Mutant operators are categorized using syntactic criteria. Such a classification is expected to be useful for an implementor of a mutation based testing system.

Another classification, useful for the tester, is based on the nature of tests that can be conducted using mutation analysis. This classification exposes the generality and completeness of mutation based testing.

Each mutant operator is introduced with illustrative examples. The rationale supporting each operator is also provided. An appendix provides a cross-reference of all mutant operators for ease of referencing.

The design described here is the result of long deliberations amongst the authors of this report in which several aspects of the C language and program development in C were examined. We intend this report to serve as a manual for the C mutant operators for researchers in software testing.

1 INTRODUCTION

This report documents the classification, definition, rationale, and semantics of the mutant operators designed for the (proposed) ANSI C programming language [Kern88]. The design of mutant operators was carried out by the authors of this report over a six month period starting August, 1988.

Throughout the report, the following conventions are used: (a) keywords in C are emboldened, (ii) non-terminal symbols from the C grammar are italicized, (iii) mutant operators are emboldened and appear in upper case letters, (iv) mutant operator category names are emboldened and begin with an upper case letter followed by three lower case letters, and (v) P denotes the program under test and hence the program to be mutated. In general, P will consist of several functions that could be mutated individually or as a group depending on how much of P is being tested.

This report assumes that the reader is familiar with the underlying theory and techniques of mutation analysis. However, the next section provides an overview of mutation analysis. [DeMi79] and [DeMi87] contain the in depth background material.

The remainder of the report is organized into 13 sections. The next section provides an overview of mutation analysis. Section 3 enumerates the reasons responsible for the mutant operators reported here. Section 4 presents a classification of the mutant operators. We expect such a classification to be useful for the tool designer[1] and for the software tester.

Section 5 consolidates the naming conventions used while naming the large number of mutant operators. The conventions are designed to simplify the expansion of the four letter mnemonics for all mutant operators. Not all entities in a program are subject to mutation. Section 6 lists all such entities and the rationale for not mutating them. Section 7 lists various *mutate-time* and *execute-time* optimizations that can be performed by a tool to reduce the computational requirements of a software testing experiment.

Some concepts and definitions needed to describe the mutant operators are listed in section 8. Sections 9-12 describe all the mutant operators for C. Section 13 compares the mutant operators for Fortran 77 and C. Finally, section 14 outlines the on-going and planned work in mutation based

[1]We use the term *tool* to refer to a software testing package based on mutation analysis, such as the Mothra [DeMi88] system.

testing of large[2] C programs.

Three appendices at the end respectively provide index to mutant operators, classification of mutant operators, and a revision history of this report.

2 AN OVERVIEW OF MUTATION BASED TESTING

In this section we provide an overview of mutation based testing. For details see [DeMi79].

Testing techniques can be classified into two categories : functional testing and functional analysis [Howd87]. Functional testing implies the existence of an input-output oracle. Such an oracle determines if the output of a function f on a test case t is correct or not. Control flow coverage and fault based techniques fall into the functional testing category. Functional analysis implies the existence of a sequence oracle. Such an oracle determines if a given sequence of actions takes place when f is executed on t.

Mutation analysis is a fault based technique. It can be classified into the functional testing category. It assumes the existence of a class F of faults. It provides a set O of operators, better known as *mutant* operators, that model one or more of the faults in F. For example, use of a wrong variable is a fault. The *scalar variable reference replacement* set of operators, described in section 11.1, model this fault.

A mutant operator is *applied* on the program P under test. Such an application transforms P into a similar, though a different program, known as a *mutant*. In general, one application of a mutant operator can generate more than one mutant. If P contains several entities that are in the domain of a mutant operator, then the operator is applied to each such entity, one at a time. Each application generates a distinct mutant. As an example, consider the mutant operator that deletes a statement from P. All statements in P are in the domain of this operator. When applied, it will generate as many mutants as there are statements in P. Each mutant will have all statements in P except the one deleted by the mutant operator. Such mutants can be considered to be *fault induced* versions of P.

Certain mutants are *instrumented* versions of P. The instrumentation is designed to reveal some kind of coverage. For example, a mutant operator that provides domain coverage for variables,

[2]We are referring to C codes that are over a million lines of code in length.

generates one mutant for each occurrence of a scalar reference. When executed, this mutant informs the tester whether or not the desired domain was covered for that reference.

Once generated, a mutant M is executed against a suite T of test cases. For a fault induced mutant, if for any $t \in T$, the output of M differs from that of P, we say that M is *killed*. An instrumented mutant is killed when the *trap* function, inserted into P by the mutant operator, terminates mutant execution. The *trap* functions are described in section 8.1. A mutant not killed by any $t \in T$ is considered to be *live*. A live mutant M implies that either (a) T needs to be augmented with additional test cases that can kill M or (b) M is *equivalent* to P. T is considered *adequate* if it is able to kill each of those mutants that are not equivalent to P. Thus, mutation analysis is often referred to as a technique for determining the adequacy of a test suite [DeMM87].

Several control flow and other coverage techniques popular amongst software testers, are subsumed by mutation analysis. For example, statement coverage, branch coverage, and domain coverage are provided by mutation analysis. The **STRP**, **STRI**, and **VDOM** operators described in this report, provide statement, branch, and domain coverage for C programs. Several coverage operators provide partial path analysis. For example, the **SMTT** operator provides iteration coverage by ensuring that each loop in a C program is executed at least n times, where n is a tester defined parameter.

As mentioned earlier, a mutant is generated by the application of one mutant operator on one entity in its domain in P. Mutants so generated are also known as *first order* mutants. In principle, however, it is possible to apply simultaneously more than one mutant operator or one mutant operator on more than one entity in its domain. Such an application will generate *k-order* mutants for $k > 1$. However, past experimental [Budd80] work has shown that such mutants do not provide any significant advantage with respect to the construction of a better test suite T. Further, the generation and execution of such mutants can be computationally expensive. Thus, mutation analysis is generaly accredited with the generation of first-order mutants. If higher order mutants are generated, then mutation analysis subsumes path analysis.

A tool based on mutation analysis, such as *Mothra* [DeMi88], automates several of the tasks implicit in the above description. For example, *Mothra* performs the tasks associated with mutant generation, mutant execution, live/kill analysis, test case management, and automatic test case generation.

3 THE RAISON D´ÊTRE OF A MUTANT OPERATOR

Errors could be introduced into a program in a variety of ways. Assuming the validity of the *competent programmer hypothesis* [DeMi79], we can conclude that errors are introduced into a program through syntactic aberrations, also known as *faults*. These aberrations alter the semantics of the program so that it fails to perform the desired input-output mapping. Note that we are not concerned with syntactic aberrations that result in syntax errors. Such errors are obviously caught by the compiler. Instead, we are concerned with aberrations that cause a change in the intended semantics of the program, thereby inducing an incorrect input-output behavior. A few examples of such errors appear below.

1. Incorrect use of a variable name in a specific context, e.g. using the statement next_line = 1 instead of new_line = 1, where *next_line* and *new_line* denote variables, with similar though different meanings, declared in the program.

2. Using an incorrect relational operator in a loop condition, e.g. using the loop formulation `while` (dosage > max-value) instead of `while` (dosage < max-value).

3. Misplacing a statement, e.g. using the sequence:

   ```
   x_satellite_position = x_satellite_position + x_shift
   x_satellite_position = x_satellite_position - x_shift
   break
   ```

   instead of the sequence:

   ```
   x_satellite_position = x_satellite_position + x_shift
   break
   x_satellite_position = x_satellite_position - x_shift
   ```

The intent of software testing is to reveal as many such syntactic aberrations as feasible in a program. Mutation analysis aids in this process by identifying the common syntactic aberrations and using *mutant operators* to model them. The identification itself is carried out primarily by using past programming experience.

A mutant operator mutates *one* syntactic entity of a program. Further, only one mutant operator is applied at a time to the program under test. The intent of a mutant operator is to make simple syntactic changes. Mutation analysis as a testing technique hopes to reveal errors in the program by showing that either the syntactic change that has been induced by a mutant operator is incorrect or the syntactic entity that is mutated is the incorrect one. Thus, mutation analysis helps in both the revelation of errors and in establishing the confidence of the tester in the program.

In general, it is difficult to show how a mutant operator can help reveal an error in a program. On the other hand, a mutant operator encourages the tester to construct test data that reveals that the syntactic change induced is indeed incorrect. If the programmer fails to do so, or fails to show that the syntactic change generates an equivalent mutant, then the existence of an error becomes highly probable.

While designing the mutant operators for C, we had the following goals in view:

1. A mutant operator should cause *single-step* changes. Thus, only simple faults are induced. Transforming

   next_line = next_line + 1

   to

   next_line = next_line - 1

   is a single-step change carried out by replacing + by −. However, transforming

   next_line = next_line + 1

   to

new_line = new_line + 1

is a two step change as two instances of the syntactic entity next_line are replaced.

2. A mutant operator should generate a syntactically correct program. This ensures that mutants can be compiled and executed.

3. Mutant operators should be designed and classified so that the tester using a tool based on such operators, can selectively apply them. This ensures that the tester has complete control over the organization and the computational requirement of the test.

   For example, according to the current classification, a tester can decide to mutate only binary arithmetic operators, such as + and –, in the program under test. However, if there was only one mutant operator that would systematically mutate all operators in the program under test, such a selective application would not be possible.

4 MUTANT OPERATOR CLASSIFICATION

To aid in the understanding, documentation, and use during testing, mutant operators have been classified using a hierarchical structure. The structure is syntax-directed. It is intended for the implementors of a tool that uses the mutant operators described in this report.

Appendix B exhibits the complete syntax-directed classification. According to this classification, each mutant operator belongs to one of the following categories:

1. statement mutations,
2. operator mutations,
3. variable mutations, and
4. constant mutations.

Mutant operators in these categories are designed to model errors made by programmers in:

1. selection of identifiers and constants while formulating expressions;

2. composition of expression functions; and

3. composition of functions using iterative and conditional statements.

Note that the faults modeled by mutant operators belong to both the *commission* and *omission* categories first proposed by Basili [Basi84]. For example, mutant operators in the **Vsrr** category model some faults that belong to the commission category. Mutant operator **OCNG** models a fault that belongs to the omission category.

Another classification, useful for a tester, also appears in Appendix B. According to this classification, mutant operators are designed either for providing coverage or for inducing faults. Coverage operators aid the tester in obtaining statement, iteration, branch, and domain coverage. A combination of these operators provide partial path coverage. Fault inducing operators induce possible faults in the program under test. Faults in statements, operators, variables, and constants are modeled by such operators.

This classification reveals the fact that mutation analysis encompasses several other commonly used testing techniques such as statement coverage, branch coverage, and path analysis.

5 NAMING CONVENTIONS

In this section we describe the naming conventions for mutant operator sub-categories and mutant operators. These naming conventions have been set up to provide easy elicitation of the function of a mutant operator from its mnemonic. Due to a plethora of mutant operators in C, it has not been possible to adhere to 3-letter mnemonics used for Fortran 77 mutant operators. Instead, a 4-letter mnemonic has been developed for each mutant operator.

5.1 General

For each mutant operator we distinguish between its category and subcategory. There are four syntax-directed categories mentioned above. These are denoted by letters **S**, **O**, **V**, and **C** for, respectively, **s**tatement, **o**perator, **v**ariable, and **c**onstant level mutations. The remaining three letters either serve as a mnemonic for the mutant operator or are further classified, depending on the category and subcategory. Figs. 2–4 exhibit this classification.

Figure 1: Naming convention for categories.

Figure 2: Naming convention for statement level mutation operators.

A subcategory of mutant operators also has a 4-letter mnemonic. The first letter denotes the category to which it belongs. The remaining three letters serve as the mnemonic for the subcategory. To enable easy distinction between subcategory and mutant operator mnemonics, the last three letters of a subcategory mnemonic are in lower case letters. The operator mutations have a more elaborate naming convention below.

5.2 Naming of Binary Operator Mutations

The binary operator mutations are described in section 10. As shown in Fig. 3, each mutant operator has a 4-letter name starting with the letter **O**. The naming conventions for the remaining three letters

Figure 3: Naming convention for C-operator level mutation operators.

Figure 4: Naming convention for variable and constant level mutation operators.

are as follows:

- *Non-assignment type:* The mutant name has the structure: **OXYN**. Here X and Y could be any of **A**, **L**, **R**, or **B** for, respectively, arithmetic, logical, relational, and bitwise operators. For mutant operators that belong to the **Ocor** category, X = Y. Thus, for example, **OBBN** mutates operators amongst the set of **b**itwise operators defined in Table 2. As another example, **OABN** mutates operators in the **a**rithmetic operators set with those in the **b**itwise operators set.

- *Assignment type:* The mutant name has the structure: **OXYA**. For mutant operators that belong to **Ocor** category, X=Y. Here X and Y can be **A**, **S**, or **B** for, respectively, arithmetic, bitwise, plain, and shift assignment operators. See Table 2 for a list of these operators.

  For the **Oior** category, X and Y could also be **E**. For example, **OBBA** mutates C operators in the **b**itwise assignment operator set. **OASA** mutates elements of **a**rithmetic assignment set to elements of **s**hift assignment set. **OSEA** mutates C operators belonging to the set of **s**hift assignment operators to the plain assignment operator (=).

6 WHAT IS NOT MUTATED ?

Every mutation operator has a possibly infinite domain on which it operates. The domain itself consists of instances of syntactic entities, that appear within the program under test, mutated by the operator. For example, the mutation operator that replaces a `while` statement by a `do-while` statement has all instances of the `while` statements in its domain. This example, however, illustrates a situation in which the domain is known.

Consider a C function having only one declaration statement `int` x, y, z. What kind of syntactic aberrations can one expect in this declaration ? One aberration could be that though the programmer intended z to be a real variable, it was declared as an integer. Certainly, a mutation operator can be defined to model such an error. However, the list of such aberrations is possibly infinite and, if not impossible, difficult to enumerate. The primary source of this difficulty is the infinite set of type and identifier associations to select from. Thus, it becomes difficult to determine the domain for any mutant operator that might operate on a declaration.

The above reasoning leads us to treat declarations as *universe defining* entities in a program. The universe defined by a declaration, such as the one mentioned above, is treated as a collection of facts. Thus, the declaration `int` x, y, z states three facts, one for each of the three identifiers. Once we regard declarations to be program entities that state facts, we cannot mutate them because we have assumed that there is no scope for any syntactic aberration. With this reasoning as the support, we decided not to mutate any declaration in a C program. We expect that errors in declarations would manifest through one or more mutants.

Following is the complete list of entities that are not mutated:

- declarations,
- the address operator (&),
- format strings in input-output functions,
- function declaration headers,
- control line,
- function name[3] indicating a function call, and
- preprocessor conditionals.

[3]Note that actual parameters in a call are mutated, but the function name is not. This implies that I/O function names such as scanf, printf, open, etc. are not mutated.

7 OPTIMIZATIONS

In certain situations, it is possible that two mutant operators generate mutants that are equivalent to each other. A simple example is provided by the **VTWD** operator described in section 11.8. When applied to the expression $(a + b)$, it will mutate a to $a + 1$ and b to $b + 1$. This generates two equivalent mutants. We assume that the mutant generation tool will detect this equivalence and generate only one of these mutants.

As another example, consider a C function containing the following statement[4]:

```
  ⋮
   if (x < 0 )
     x = x-y;                                                    F1
  ⋮
```

One of the mutant operators, namely **SSDL** defined in section 9.3, will mutate the above statement to the following:

```
  ⋮
   if (x < 0 )
     ;                                                           M1
  ⋮
```

and

```
  ⋮
     ; /* Entire if statement replaced by a null statement. */   M2
  ⋮
```

The mutants so generated are equivalent as can be concluded by examining the two mutated statements above. It is expected that a tool that implements various mutant operators will attempt not generate a mutant that is equivalent to the one already generated.

[4]Program fragments used in examples, are numbered sequentially as F1, F2,... . Mutants of program fragments are numbered M1, M2, ...

Table 1: List of Functions Introduced by Mutant Operators

Function name	Number of arguments	Purpose	Introduced by
break_out_to_level_n	1	To terminate the execution of each one of the n loops immediately enclosing this function.	**SBRn**
continue_out_to_level_n	1	To terminate the execution of all immediately enclosing $(n-1)$ loops and resume the next iteration of the loop that nests this function n levels.	**SCRn**
trap on domain functions	2	To terminate the execution if the domain of the argument satisfies a given condition. See section 9.2 and Table 6 on page 60 for details.	**VABS**
trap_on_case	2	If the first argument is equal to the second, then the mutant is killed, otherwise it returns the value of the first argument.	**SSWM**
trap_on_statement	0	To terminate mutant execution	**STRP**
false_after_n^{th}_loop_iteration	1	To force the loop *body* to execute at most n times	**SMTC**
trap_after_n^{th}_loop_iteration	1	Returns true for each of the first n iterations. Terminates program execution at the beginning of the $(n+1)^{th}$ iteration.	**SMTT**

8 CONCEPTS AND DEFINITIONS

8.1 Functions Introduced by Mutant Operators

Several mutant operators introduce a function into the source program. A list of all such functions appears in Table 1. We assume that the source program contains no functions with the same name as the function introduced by the mutant operator.

8.2 Range of Applicability

In this report, the well accepted definition of a C program is assumed. According to this definition, a C program consists of a collection of functions and variables. These may be grouped into different files.

We define the *range of applicability* of a mutant operator, hereafter referred to as RAP, to be a subset of all functions in the program under test. Such a RAP is defined, under directions from the tester, by the tool. The grouping of functions into different files does not affect the RAP.

A mutant operator is considered to be applicable only to *all* the program entities in its domain within the RAP. For example, the mutant operator **SSDL** that delete statements from a program, will apply to all statements inside functions that are in the RAP. It is possible for different mutant operators to have different RAPs. However, this fact does not affect the definition of a mutant operator in any way.

8.3 Linearization

In C, the definition of statement is recursive. For the purpose of understanding various mutant operators in the statement mutations category, we introduce the concept of *linearization* and *reduced linearized sequence.*

Let S denote a syntactic construct that can be parsed as a C *statement.* Note that *statement* is a syntactic category in C. For an iterative or selection statement denoted by S, ${}^{c}S$ denotes the condition controlling the execution of S . If S is a `for` statement, then ${}^{e}S$ denotes the expression that is executed immediately after one execution of the loop body and just before the next iteration of the loop body, if any, is about to begin. Again, if S is a `for`, then ${}^{i}S$ denotes the initialization expression that is executed exactly once for each execution of S. If the controlling condition is missing, then ${}^{c}S$ defaults to `true`.

Using the above notation, if S is an `if` statement, we shall refer to the execution of S in an execution sequence as ${}^{c}S$. If S denotes a `for` statement, then in an execution sequence we shall refer to the execution of S by one reference to ${}^{i}S$, one or more references to ${}^{c}S$, and zero or more references to ${}^{e}S$. If S is a compound statement, then referring to S in an execution sequence merely refers to any storage allocation activity.

Example 1

Consider the following `for` statement:

```
for (m=0, n=0; isdigit(s[i]); i++)
  n =10* n +(s[i]) - '0');
```

Denoting the above `for` statement by S, we get,

^{i}S: m=0, n=0

^{c}S: isdigit(s[i]), and

^{e}S: i++.

If S denotes the following `for` statement,

```
  for (; ;){
⋮
  }
```

then we have,

^{i}S: ; (the null expression-statement),

^{c}S: `true`, and

^{e}S: ; (the null expression-statement).

■

Let T_f and T_S, respectively, denote the parse trees of function f and statement S. A node of T_S is said to be *identifiable* if it is labeled by any one of the following syntactic categories:

- statement
- labeled statement

- expression statement
- compound statement
- selection statement
- iteration statement
- jump statement

A *linearization* of S is obtained by traversing T_S in preorder and listing, in sequence, only the identifiable nodes of T_S. Fig. 5 provides examples of linearized statements.

For any X, let X_j^i, $1 \leq j \leq i$ denote the sequence $X_j \; X_{j+1} \; \ldots \; X_{i-1} \; X_i$. Let $S_L = S_1^l, : l \geq 0$ denote the linearization of S. If $S_i \; S_{i+1}$ is a pair of adjacent elements in S_L such that S_{i+1} is the direct descendent of S_i in T_S and there is no other direct descendent of S_i, then $S_i S_{i+1}$ is considered to be a *collapsible* pair with S_i being the *head* of the pair. A *reduced linearized sequence* of S, abbreviated as RLS, is obtained by recursively replacing all collapsible elements of S_L by their heads. The RLS of a function is obtained by considering the entire body of the function as S and finding the RLS of S. The RLS, obtained by the above method, will yield a statement sequence in which the indices of the statements are not increasing in steps of 1. We shall always simplify the RLS by renumbering its elements, so that for any two adjacent elements $S_i \; S_j$, we have $j = i + 1$.

We shall refer to the RLS of a function f and a statement S by $RLS(f)$ and $RLS(S)$, respectively. Fig. 5 lists reduced linearization sequences of sample statements.

$RLS(f)$ is of the form $S_1 \; S_2 \; \ldots S_n \; R,\; n \geq 0$, where R is either a `return` statement or an implicit return implied by the closing brace in the definition of f. The S_i, $1 \leq i \leq n$ and R are referred to as the *elements* of RLS. The number of elements in RLS is its length and is denoted by $l(RLS)$. We shall use an RLS to define mutant operators and describe execution sequences.

8.4 Execution Sequence

Though most mutant operators are designed to simulate simple faults, the expectation of mutation based testing is that such operators will eventually reveal one or more errors in the program. In this section we provide some basic definitions that are useful in understanding such operators and their dynamic effects on the program under test.

Figure 5: Linearization examples. (+ denotes one or more derivation steps.)

Figure 5: Linearization examples(contd.)(+ denotes one or more derivation steps.).

When f executes, the elements of $RLS(f)$ will be executed in an order determined by the test case and any path conditions in $RLS(f)$. Let $E(f,t) = s_1^m$, $m \geq 1$ be the execution sequence of $RLS(f) = S_1^n\ R$ for test case t, where s_j, $1 \leq j \leq m-1$ is any one of S_i, $1 \leq i \leq n$ and S_i is not a **return** statement. We assume that f terminates on t. Thus, $s_m = R'$, where R' is R or any other return statement in $RLS(f)$.

Any proper prefix s_1^k, $0 < k < m$ of $E(f,t)$, where $s_k = R'$, is a *prematurely terminating execution sequence* (subsequently referred to as $PTES$ for brevity) and is denoted by $E^p(f,t)$. s_{k+1}^m, is known as the *suffix* of $E(f,t)$ and is denoted by $E^s(f,t)$. $E^l(f,t)$ denotes the last statement of the execution sequence of f. If f is terminating, $E^l(f,t)$=**return**.

Let $E_1 = S_i^j$ and $E_2 = Q_k^l$ be two execution sequences. We say that E_1 and E_2 are *identical* if and only if $i = k, j = l$, and $S_q = Q_q$, $i \leq q \leq j$. As a simple example, if f and f' consist of one assignment each, namely, $a = b + c$ and $a = b - c$, respectively, then there is no t for which $E(f,t)$ and $E(f',t)$ are identical. It must be noted that the output generated by two execution sequences may be the same even though the sequences are not identical. In the above example, for any test case t that has $c = 0$, $P_f(t) = P_{f'}(t)$.

Example 2

Consider the function trim defined below.

```
        /* This function is from p 65 of [Kern88]. */

        int trim(char s[] )
S1      {
          int n;                                                     F 2
S2        for (n = strlen(s)-1; n >= 0; n--)
S3          if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
S4            break;
S5        s[n+1] = '\0';
S6        return n;
        }
```

We have $RLS(f) = S_1\,S_2\,S_3\,S_4\,S_5\,S_6$. Let the test case t be such that the input parameter s evaluates to ab (space follows b), then the execution sequence $E(f,t)$ is: $S_1\,{}^iS_2\,{}^cS_2\,{}^cS_3\,{}^cS_2\,{}^cS_3\,S_4\,S_5\,S_6$. $S_1\,S_2$ is one prefix of $E(f,t)$ and $S_4\,S_5\,S_6$ is one suffix of $E(f,t)$. Note that there are several other prefixes and suffixes of $E(f,t)$. $S_1\,{}^iS_2\,{}^cS_2\,S_6$ is a proper prefix of $E(f,t)$. ■

Analogous to the execution sequence for $RLS(f)$, we define the execution sequence of $RLS(S)$ denoted by $E(S,t)$ with $E^p(S,t)$, $E^s(S,t)$, and $E^l(S,t)$ corresponding to the usual interpretation.

The composition of two execution sequences $E_1 = p_1^k$ and $E_2 = q_1^l$ is $p_1^k\,q_1^l$ and is written as $E_1 \circ E_2$. The conditional composition of E_1 and E_2 with respect to condition c, is written as $E_1 \,|_c\, E_2$. It is defined as:

$$E_1 \,|_c\, E_2 = \begin{cases} E_1 & \text{if } c \text{ is false,} \\ E_1 \circ E_2 & \text{otherwise.} \end{cases}$$

In the above definition, condition c is assumed to be evaluated after the entire E_1 has been executed. Note that $\circ$ has the same effect as $|_{\texttt{true}}$. $\circ$ associates from left to right and $|_c$ associates from right to left. Thus, we have:

$$\begin{aligned} E_1 \circ E_2 \circ E_3 &= (E_1 \circ E_2) \circ E_3 \\ E_1 \,|_{c_1}\, E_2 \,|_{c_2}\, E_3 &= E_1 \,|_{c_1}\, (E_2 \,|_{c_2}\, E_3) \end{aligned}$$

$E(f,*)$ $(E(S,*))$ denotes the execution sequence of function f (statement S) on the *current* values of all the variables used by f (S). We shall use this notation while defining execution sequences of C functions and statements.

Let S, S_1, and S_2 denote a C statement other than `break`, `continue`, `goto`, and `switch`, unless specified otherwise.. The following rules can be used to determine execution sequences for any C function.

R 1 E({ },t) is the null sequence.

R 2 $E(\{\ \},t) \circ E(S,t) = E(S,t) = E(S,t) \circ E(\{\ \})$

R 3

$$E(\{\ \},t) \,|_c\, E(S,t) = |_c\, E(S,t) = \begin{cases} \text{null sequence} & \text{if } c \text{ is false,} \\ E(S,t) & \text{otherwise.} \end{cases}$$

R 4 If S is an *assignment-expression*, then $E(S,t) = S$.

R 5 For any statement S, $E(S,t) = RLS(S)$, if $RLS(S)$ contains no statements other than zero or more assignment-expressions. If $RLS(S)$ contains any statement other than the assignment-expression, the above equality is not guaranteed due to the possible presence of conditional and iterative statements.

R 6 If $S = S_1 \,;\, S_2$; then $E(S,t) = E(S_1,t) \circ E(S_2,*)$.

R 7 If $S =$ `while` (c) S_1', then

$$E(S,t) =|_c (E(S_1',*) \circ E(S,*))$$

If $RLS(S) = S_1^n$, $n > 1$, and $S_i =$ `continue`, $1 \leq i \leq n$, then

$$E(S,t) =|_c E(S_1^i,*) \circ (|_{(E^l(S_1^i,*) \neq \texttt{continue})} E(S_{i+1}^n,*)) \circ E(S,*)$$

If $RLS(S) = S_1^n$, $n > 1$, and $S_i =$ `break`, $1 \leq i \leq n$, then

$$E(S,t) =|_c E(S_1^i,*) \circ (|_{(E^l(S_1^i,*) \neq \texttt{break})} (E(S_{i+1}^n) \circ E(S,*)))$$

R 8 If $S =$ `do` S_1`while`(c); then

$$E(S,t) = E(S_1,t) \,|_c\, E(S,*)$$

If $RLS(S)$ contains a `continue`, or a `break`, then its execution sequence can be derived using the method indicated for the `while` statement.

R 9 If $S =$ `if` (c) S_1, then

$$E(S,t) =|_c E(S_1,*).$$

R 10 If $S =$ `if` (c) S_1 `else` : S_2, then

$$E(S,t) = \begin{cases} E(S_1,t) & \text{if } c \text{ is true,} \\ E(S_2,t) & \text{otherwise.} \end{cases}$$

Example 3

Consider S_3, the `if` statement, in F 2. We have $RLS(S_3) = S_3\ S_4$. Assuming the test case of the Example on page 19 and $n = 3$, we get $E(S_3, *) = {}^cS_3$. If $n = 2$, then $E(S_3, *) = {}^cS_3\ S_4$.

Similarly, for S_2, the `for` statement, we get $E(S_2, *) = {}^iS_2\ {}^cS_2\ {}^cS_3\ {}^cS_2\ {}^cS_3\ S_4$. For the entire function body, we get $E(f, t) = S_1\ E(S_2, *) \circ E(S_5, *) \circ E(S_6, *)$. ■

8.5 Effect of An Execution Sequence

As before, let P denote the program under test, f, a function in P, that is to be mutated, and t a test case. Assuming that P terminates, let $P_f(t)$ denote the output generated by executing P on t. The subscript f with P is to emphasize the fact that it is the function f that is being mutated.

We say that $E(f, *)$ has a distinguishable effect on the output of P, if $P_f(t) \neq P_{f'}(t)$, where f' is a mutant of f. We consider $E(f, *)$ to be a *distinguishing execution sequence* (hereafter abbreviated as DES) of $P_f(t)$ with respect to f'.

Given f and its mutant f', for a test case t to kill f', it is necessary, but not sufficient, that $E(f, t)$ be different from $E(f', t)$. The sufficiency condition is that $P_f(t) \neq P_{f'}(t)$ implying that $E(f, t)$ is a DES for $P_f(t)$ with respect to f'.

While describing the mutant operators, we shall often use DES to indicate when a test case is sufficient to distinguish between a program and its mutant. Examining the execution sequences of a function, or a statement, can be useful in constructing a test case that kills a mutant.

Example 4

To illustrate the notion of the effect of an execution sequence, consider the function *trim* defined in F 2. Suppose that the output of interest is the string denoted by s. If the test case t is such that s consists of the three characters a, b, and space, in that order, then $E(trim, t)$ generates the string ab as the output. As this is the intended output, we consider it to be correct.

Now suppose that we modify f by mutating S_4 in F 2 to `continue`. Denoting the modified function by $trim'$, we get:

$$E(trim', t) = S_1\ {}^iS_2\ {}^cS_2\ {}^cS_3\ {}^eS_2\ {}^cS_3\ {}^eS_2\ {}^cS_3\ S_4\ {}^eS_2\ {}^cS_3\ S_4\ {}^eS_2\ S_5\ S_6 \quad (1)$$

The output generated due to $E(trim', t)$ is different from that generated due to $E(trim, t)$. Thus, $E(trim, t)$ is a DES for $P_{trim}(t)$, with respect to the function $trim'$. ■

DES's are essential to kill mutants. To obtain a DES for a given function, a suitable test case needs to be constructed such that $E(f, t)$ is a DES for $P_f t$ with respect to f'.

8.6 Global and Local Identifier Sets

For defining variable mutations in section 11, we need the concept of global and local sets, defined in this section, and global and local reference sets, defined in the next section.

Let f denote a C function to be mutated. An identifier denoting a variable, that can be used inside f, but is not declared in f, is considered global to f. Let G_f denote the set of all such global identifiers for f. Note that any *external* identifier is in G_f unless it is also declared in f. While computing G_f, it is assumed that all files specified in one or more `# include` control lines have been *included* by the C pre-processor. Thus, any global declaration within the files listed in a `# include`, also contributes to G_f.

Let L_f denote the set of all identifiers that are declared either as parameters of f or at the head of its body. Identifiers denoting functions do not belong to G_f or L_f.

In C, it is possible for a function f to have nested compound statements such that an inner compound statement S has declarations at its head. In such a situation, the global and local sets for S can be computed using the scope rules in C.

We define GS_f, GP_f, GT_f, and GA_f as subsets of G_f which consist of, respectively, identifiers declared as scalars, pointers to an entity, structures, and arrays. Note that these four subsets are pairwise disjoint. Similarly, we define LS_f, LP_f, LT_f, and LA_f as the pairwise disjoint subsets of L_f.

8.7 Global and Local Reference Sets

Use of an identifier within an expression is considered a *reference*. In general, a reference can be *multilevel* implying that it can be composed of one or more sub-references. Thus, for example, if ps is a pointer to a structure with components a and b, then in $(*ps).a$, ps is a reference and $*ps$ and $(*ps).a$ are two sub-references. Further, $*ps.a$ is a 3-level reference. At level 1, we have ps, at level 2 we have $(*ps)$, and finally at level 3 we have $(*ps).a$. Note that in C, $(*ps).a$ has the same meaning as $ps{-}{>}a$.

The global and local reference sets consist of references at level 2 or higher. Any references at level 1 are in the global and local sets defined earlier. We shall use GR_f and LR_f to denote, respectively, the global and local reference sets for function f.

Referencing a component of an array or a structure may yield a scalar quantity. Similarly, dereferencing a pointer may also yield a scalar quantity. All such references are known as *scalar* references. Let GRS_f and LRS_f denote sets of all such global and local scalar references, respectively. If a reference is constructed from an element declared in the global scope of f, then it is a global reference, otherwise it is a local reference.

We now define GS'_f and LS'_f by augmenting GS_f and LS_f as follows:

$$\begin{aligned} GS'_f &= GRS_f \cup GS_f \\ LS'_f &= LRS_f \cup LS_f \end{aligned}$$

GS'_f and LS'_f are termed as scalar global and local reference sets for function f, respectively.

Similarly, we define array, pointer, and structure reference sets denoted by, respectively, GRA_f, GRP_f, GRT_f, LRA_f, LRP_f, and LRT_f. Using these, we can construct the augmented global and local sets GA'_f, GP'_f, GT'_f, LA'_f, LP'_f, and LT'_f.

For example, if an array is a member of a structure, then a reference to this member is an array reference and hence belongs to the array reference set. Similarly, if a structure is an element of an array, then a reference to an element of this array is a structure reference and hence belongs to the structure reference set.

On an initial examination, our definition of global and local reference sets might appear to be ambiguous specially with respect to a pointer to an *entity*[5]. However, if fp is a pointer to some entity, then fp is in set GRP_f or LRP_f depending on its place of declaration. On the other hand, if fp is an entity of pointer(s), then it is in any one of the sets GRX_f or LRX_f where X could be any one of the letters A, P, or T.

[5]An *entity* in the present context can be a scalar, an array, a structure, or a pointer. Function references are not mutated.

To illustrate our definitions, consider the following external declarations for function f:

```
int i, j; char c, d; double r, s;
int *p, *q[3];
struct point {
  int x;
  int y;
};

struct rect {
  struct point p1;
  struct point p2;                                                    F3
};

struct rect screen;
struct key {
  char * word;
  int count;
} keytab [ NKEYS ];
```

The global sets corresponding to the above declarations are:

$$\begin{aligned} G_f &= \{i, j, c, d, r, s, p, q, screen, keytab\} \\ GS_f &= \{i, j, c, d, r, s\} \\ GP_f &= \{p\} \\ GT_f &= \{screen\} \\ GA_f &= \{q, keytab\} \end{aligned}$$

Note that structure components x, fy, *word*, and *count* do not belong to any global set. Type names, such as *rect* and *key* above, are not in any global set. Further, type names do not participate in mutation due to reasons outlined in section 6.

Now, suppose that the following declarations are within function f:

```
int fi; double fx; int *fp, int (*fpa) (20)

struct rect fr; struct rect *fprct;

int fa[10]; char *fname[nchar]
```

Then, the local sets for f are:

$$\begin{aligned} L_f &= \{fi, fx, fp, fpa, fr, fprct, fa, fname\} \\ LA_f &= \{fa, fname\} \\ LP_f &= \{fp, fpa, fprct\} \\ LS_f &= \{fi, fx\} \\ LT_f &= \{fr\} \end{aligned}$$

To illustrate reference sets, suppose that f contains the following references (the specific statement context in which these references are made is of no concern for the moment):

```
i * j + fi
r + s - fx + fa[i]
*p += 1
*q[j] = *p
screen.p1 = screen.p2
screen.p1.x = i
keytab[j].count = *p
p = q[i]
fr = screen
*fname[j] = keytab[i].word
fprct = &screen
```

The global and local reference sets corresponding to the above references are:

$$\begin{aligned}
GRA_f &= \{\ \} \\
GRP_f &= \{q\,[\,i\,],\ keytab\,[i\,].word,\ \&\,screen\} \\
GRS_f &= \{keytab\,[j\,].count,\ *p,\ *q\,[\,j\,],\ screen.p1.x\} \\
GRT_f &= \{keytab\,[\,i\,],\ keytab\,[\,j\,],\ screen.p1,\ screen.p2\}
\end{aligned}$$

$$\begin{aligned}
LRA_f &= \{\ \} \\
LRP_f &= \{fname\,[\,j\,]\} \\
LRS_f &= \{*fname[\,j\,],\ fa\,[\,i\,]\ \} \\
LRT_f &= \{\ \}
\end{aligned}$$

The above sets can be used to augment the local sets.

Analogous to the global and local sets of variables, we define global and local sets of constants: GC_f and LC_f. GC_f is the set of all constants global to f. LC_f is the set of all constants local to f. Note that a constant can be used within a declaration or in an expression.

We define GCI_f, GCR_f, GCC_f, and GCP_f to be subsets of GC_f consisting of only integer, real, character, and pointer constants. GCP_f consists of only `null`. LCI_f, LCR_f, LCC_f, and LCP_f are defined similarly.

9 STATEMENT MUTATIONS

In this section we shall describe each one of the mutant operators that mutate entire statements or their key syntactic elements. For each mutant operator, in this and subsequent sections, its definition and the error modeled is provided. The domain of a mutant operator is described in terms of the syntactic entity that is affected. While mentioning this syntactic entity we have used the grammar described in [Kern88]. An appendix at the end of the report provides an index for easy location of a mutant operator. This appendix also lists the domain of all mutant operators.

Note that the operator and variable mutations described in subsequent sections, also affect statements. However, they are not intended to model errors in the explicit composition of the *selection, iteration*, and *jump* statements.

9.1 Trap on Statement Execution: STRP

This operator is intended to reveal unreachable code in the program.

Each statement is systematically replaced by trap_on_statement(). When trap_on_statement is executed, mutant execution terminates. The mutant is treated as killed. For example, consider the following program fragment:

```
while (x != y)
 {
   if (x < y)                                            F4
     y -= x;
   else
     x -= y;
 }
```

When **STRP** is applied to the above statement, a total of four mutants are generated as shown in M 3, M 4, M 5, and M 6. Test cases that kill all these four mutants are sufficient to guarantee that all the four statements in F 4 have been executed at least once.

```
trap_on_statement();                                     M3

while (x != y)
 {
   trap_on_statement();                                  M4
 }

while (x != y)
```

```
  {
    if (x < y)                                           M 5
      trap_on_statement();
    else
      x -= y;
  }

while (x != y)
  {
    if (x < y)                                           M 6
      y -= x;
    else
      trap_on_statement();
  }
```

If **STRP** is used with the RAP set to include the entire program, the tester will be forced to design test cases that guarantee that all statements have been executed. Failure to design such a test set implies that there is some unreachable code in the program. Recall that in the popular testing literature, such testing is often referred to as obtaining *statement coverage* [Howd87]

9.2 Trap on *if* Condition: STRI

STRI is designed to provide branch analysis for any `if`-statements in P. When used in addition to the **STRP**, **SSWM**, and **SMTT** operators, complete branch analysis can be performed.

When applied on P, **STRI** generates two mutants for each `if` statement. For example, for the statement: `if` $(e)S$, the two mutants generated are:

```
v = e
if (trap_on_true(v)) S                                   M 7

v = e
```

`if` (trap_on_false(v)) S M 8

In the above examples, v is assumed to be a new scalar identifier not declared in P. The type of v is the same as that of e.

When trap_on_true (trap_on_false) is executed, the mutant is killed if the function argument value is true (false). If the argument value is not true (false), then the function returns false (true) and the mutant execution continues.

STRI encourages the tester to generate test cases so that each branch specified by a `if` statement in P, is exercised at least once.

For an implementor of a mutation-based tool, it is useful to note that **STRP** provides partial branch analysis for `if` statements. For example, consider a statement of the form: `if` (c) S_1 `else` S_2. The **STRP** operator will have this statement replaced by the following statements to generate two mutants:

- `if` (c) trap_on_statement() `else` S_2
- `if` (c) S_1 `else` trap_on_statement()

Killing both these mutants implies that both the branches of the `if-else` statement have been traversed. However, when used with a `if` statement without an *else* clause, **STRP** may fail to provide coverage of both the branches.

9.3 Statement Deletion: SSDL

SSDL is designed to show that each statement in P has an effect on the output. SSDL encourages the tester to design a test set that causes all statements in the RAP to be executed and generates outputs that are different from the program under test.

When applied on P, **SSDL** systematically deletes each statement in $RLS(f)$. For example, when **SSDL** is applied on F 4, four mutants are generated as shown in M 9, M 10, M 11, and M 12.

```
;                                                        M 9

while (x != y)
  {
                                                         M 10
  }

while (x != y)
  {
    if (x < y)                                           M 11
      ;
    else
      x -= y;
  }

while (x != y)
  {
    if (x < y)                                           M 12
      y -= x;
    else
      ;
  }
```

To maintain the syntactic validity of the mutant, **SSDL** ensures that the semicolons are retained when a statement is deleted. In accordance with the syntax of C, the semicolon appears only at the end of (i) *expression-statement* and (ii) `do-while` *iteration-statement*. Thus, while mutating an *expression-statement*, **SSDL** deletes the optional *expression* from the statement, retaining the semicolon. Similarly, while mutating a `do-while` *iteration-statement*, the semicolon that terminates this statement is retained. In other cases, such as the *selection-statement*, the semicolon automatically gets retained as it is not a part of the syntactic entity being mutated[6].

[6]We considered naming the statement deletion operator *statement replacement by null statement*. As *null statement*

9.4 `return` Statement Replacement: SRSR

When a function f executes on test case t, it is possible that due to some error in the composition of f, certain suffixes of $E(f,t)$ do not affect the output of P. In other words, a suffix may not be a DES of $P_f(t)$ with respect to f' obtained by replacing an element of $RLS(f)$ by a **`return`**. The **SRSR** operator models such errors.

If $E(f,t) = s_1^m\ R$, then there are $m+1$ possible suffixes of $E(f,t)$. These are shown below:

$$\begin{array}{r} s_1\, s_2\ \ldots\ s_{m-1}\, s_m\ R \\ s_2\ \ldots\ s_{m-1}\, s_m\ R \\ s_{m-1}\, s_m\ R \\ \vdots \\ R \end{array}$$

In case f consists of loops, m could be made arbitrarily large by manipulating the test cases. The **SRSR** operator creates mutants that generate a subset of all possible $PMES$'s of $E(f,t)$.

Let $R_1, R_2, \ldots, R_k$ be the k **`return`** statements in f. If there is no such statement, a parameterless **`return`** is assumed to be placed at the end of the text of f. Thus, for our purpose, $k \geq 1$. The **SRSR** operator will systematically replace each statement in $RLS(f)$ by each one of the k **`return`** statements. The **SRSR** operator encourages the tester to generate at least one test case that ensures that $E^s(f,t)$ is a DES for the program under test.

Example 5

Consider the following function definition:

```
/* This is an example from p 69 of [Kern88].*/

int strindex(char s[ ], char t[ ])
```

is not a syntactic category (a non-terminal or a terminal) in C grammar, we decided against it.

```
{
    int i, j, k ;
    for (i = 0; s[i] != '\0'; i++){
      for (j=i; k=0; t[k] != '\0' && s[j] == t[k] ; j++ ; k++)
        ;                                                                   F 5
      if (k>0 && t[k] == '\0')
        return i;
    }
    return -1;
}
```

The above function will generate a total of six mutants, two of which are M 13 and M 14.

```
int strindex(char s[ ], char t[ ])
{
    int i, j, k ;

/* The outer for statement replaced by return i.

    return i;    ← /* Mutated statement. */                             M 13

    return -1;
}

/* This mutant has been obtained by replacing
    the inner for by return -1.

int strindex(char s[ ], char t[ ])
{
    int i, j, k ;
    for (i = 0; s[i] != '\ 0'; i++){
      return -1;    ← /* Mutated statement. */                          M 14
```

```
        if (k>0 && t[k] == '\ 0')
        return i;
    }
    return -1;
}
```

Note that both M 13 and M 14 generate the shortest possible $PMES$ for f. ■

9.5 goto Label Replacement: SGLR

In a function f, the destination of a `goto` may be incorrect. Altering this destination is expected to generate an execution sequence different from $E(f, t)$.

Suppose that `goto` L and `goto` M are two `goto` statements in f. We say that these are two *distinct* `goto`'s if L and M are different labels. Let `goto` l_1, `goto` l_2, ..., `goto` l_n be n distinct `goto` statements in f. The **SGLR** operator systematically mutates label l_i in `goto` l_i to $(n-1)$ labels $l_1, l_2, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$. If $n = 1$, no mutants are generated by **SGLR**.

9.6 `continue` Replacement by `break`: SCRB

A `continue` statement terminates the *current* iteration of the immediately surrounding loop and initiates the *next* iteration. Instead of the `continue`, the programmer might have intended a `break` that forces the loop to terminate. This is one error that **SCRB** models. Incorrect placement of `continue` is another error that **SCRB** expects to reveal.

SCRB replaces the `continue` statement by `break`.

If S denotes the innermost loop that contains the `continue` statement, then the **SCRB** operator encourages the tester to construct a test case t to show that $E(S, *)$ is a DES for $P_S(t)$ with respect to the mutated S. ■

9.7 `break` Replacement by `continue`: SBRC

Using `break` instead of a `continue` or misplacing a `break` are the two errors modeled by **SBRC**.

The `break` statement is replaced by `continue`. If S denotes the innermost loop containing the `break` statement, then SBRC encourages the tester to construct test data t to show that $E(S,t)$ is a DES for $P_S(t)$ with respect to S', where S' is a mutant of S.

9.8 Break Out to nth Enclosing Level: SBRn

Execution of a `break` inside a loop forces the loop to terminate. This causes the resumption of execution of the outer loop, if any. However, the condition that caused the execution of `break`, might be intended to terminate the execution of the immediately enclosing loop, or in general, the n^{th} enclosing loop. This is the error modeled by **SBRn**.

Let a `break` (or a `continue`) statement be inside a loop that is nested n levels deep.[7] The **SBRn** operator systematically replaces `break` (or `continue`) by the function break_out_to_level-n(j), for $2 \leq j \leq n$. When a **SBRn** mutant executes, the execution of the mutated statement causes the loop, inside which the mutated statement is nested, and the j enclosing loops, to terminate.

Let S' denote the loop immediately enclosing a `break` or a `continue` statement and nested n, $n > 0$, levels inside the loop S in function f. The **SBRn** operator encourages the tester to construct a test case t to show that $E^s(S,t)$ is a DES of f with respect to $P_f(t)$ and the mutated S. From the execution sequence construction rules listed in section 8.4. The exact expression for $E(S,t)$ can be derived for f and its mutant.

The **SBRn** operator has no effect on:

- `break` or `continue` statements that are nested only one level deep.
- A `break` intended to terminate the execution of a `switch` statement. Note that a `break` inside a loop nested in one of the cases of a `switch`, is subject to mutation by **SBRn** and **SBRC**.

9.9 Continue Out to nth Enclosing Level: SCRn

This operator is similar to **SBRn**. It replaces a nested `break` or a `continue` by the function continue_out_to_level_n(j), $2 \leq j \leq n$.

The **SCRn** operator has no effect on:

[7] A statement with only one enclosing loop is considered to be nested one level deep.

- `break` or `continue` statements that are nested only one level deep.
- A `continue` intended to terminate the execution of a `switch` statement. Note that a `continue` inside a loop nested in one of the cases of a `switch` is subject to mutation by **SCRn** and **SCRB**.

9.10 `while` Replacement by `do-while`: SWDD

Though a rare occurrence, it is possible that a `while` is used instead of a `do-while`. The **SWDD** operator models this error.

The `while` statement is replaced by the `do-while` statement.

Example 6

Consider the following loop:

```
/* This loop is from p 69 of [Kern88]. */

while (−−lim>0 && (c=getchar()) != EOF && c !='\n')
  s[i++]=c;                                                        F 6
```

When the **SWDD** operator is applied, it will be mutated to the loop shown in M 15. ■

```
do {
  s[i++]=c;                                                        M 15
}
  while (−−lim>0 && (c=getchar()) != EOF && c !='\n')
```

9.11 `do-while` Replacement by `while`: SDWD

The `do-while` statement may have been used in a program when the `while` statement would have been the correct choice. The **SDWD** operator models this error.

A `do-while` statement is replaced by a `while` statement.

Example 7

Consider the following `do-while` statement in P:

```
/* This loop is from p 64 of [Kern88].*/

do {
  s [ i++ ] = n % 10 + '0';                                    F 7
} while (( n /= 10) > 0);
```

It will be mutated by the **SDWD** operator to:

```
while (( n /= 10) > 0) {
  s [ i++ ] = n % 10 + '0';                                    M 16
}
```

Notice that the only test data that can kill the above mutant is one that sets n to 0 just before the loop begins to execute. This test case ensures that $E(S, *)$, S being the original `do-while` statement, is a DES for $P_S(t)$ with respect to the mutated statement, i.e. the `while` statement. ■

9.12 Multiple Trip Trap: SMTT

For every loop in P, we would like to ensure that the loop body

- C1: has been executed more than once, and
- C2: has an effect on the output of P.

The **STRP** operator replaces the loop body with the trap_on_statement. A test case that kills such a mutant implies that the loop body has been executed at least once. However, this does not ensure the two conditions mentioned above. The **SMTT** and **SMTC** operators are designed to ensure C1 and C2.

The **SMTT** operator introduces a guard in front of the loop body. The guard is a logical function named trap_after_n^{th}_loop_iteration(n). When the guard is evaluated the n^th time through the loop, it kills the mutant. The value of n is decided by the tester.

Example 8

Consider the following `for` statement:

```
/* This loop is taken from p 87 of [Kern88]. */

for (i = left+1; i <= right; i++)
  if (v [ i ] < v [ left ])                                    F 8
    swap (v, ++last, i);
```

Assuming that $n = 2$, this will be mutated by the **SMTT** operator to the loop in M 17. ■

```
for (i = left+1; i ≤ right; i++)
  if (trap_after_nth_loop_iteration)(2){
    if (v [ i ] < v [ left ])                                  M 17
      swap (v, ++last, i);
  }
```

For each loop in the program under test, the **SMTT** operator encourages the tester to construct a test case so that the loop is iterated at least twice.

9.13 Multiple Trip Continue: SMTC

An **SMTT** mutant may be killed by a test case that forces the mutated loop to be executed two times. However, it does not ensure condition C2 mentioned earlier. The **SMTC** operator is designed to ensure that C2 is satisfied.

SMTC introduces a guard in front of the loop body. The guard is a logical function named false_after_n^{th}_loop_iteration(n). During the first n iteration of the loop, false_after_n^{th}_loop_iteration()

evaluates to *true*, thus letting the loop body execute. During the $(n+1)^{th}$ and subsequent iterations, if any, it evaluates to *false*. Thus, a loop mutated by **SMTC** will iterate as many times as the loop condition demands. However, the loop body will *not* be executed during the second and any subsequent iterations.

Example 9

The loop in F 8 will be mutated by the **SMTC** operator to the loop in M 18. ■

```
for (i = left+1; i ≤ right; i++)
  if (false_after_nth_loop_iteration()){
    if (v [ i ] < v [ left ])                                        M 18
      swap (v, ++last, i);
  }
```

The **SMTC** operator may generate mutants containing infinite loops. This is specially true when the execution of the loop body affects one or more variables used in the loop condition.

For a function f, and each loop S in the RAP(f), **SMTC** encourages the tester to construct a test case t which causes the loop to be executed more than once such that $E(f,t)$ is a DES of $P_f(t)$ with respect to the mutated loop. Note that **SMTC** is stronger than **SMTT**. This implies that a test case that kills an **SMTC** mutant for statement S, will also kill the **SMTT** mutant of S.

9.14 Sequence Operator Mutation: SSOM

Use of the *comma* operator results in the left to right evaluation of a sequence of expressions and forces the value of the rightmost expression to be the *result*. For example, in the statement

$$f\,(\text{a, (b=1, b+2), c});$$

function f has three parameters. The second one has the value 3. The programmer may use the wrong sequence of expressions thereby forcing the incorrect value to be the result. The **SSOM** operator is designed to model this error.

Let $e_1, e_2, \ldots, e_n$ denote an expression consisting of a sequence of n sub-expressions[8] separated by the *comma* operator. The **SSOM** operator generates $(n-1)$ mutants of this expression by rotating left the sequence one sub-expression at a time.

Example 10

Consider the following statement:

```
/* This loop is taken from p 63 of [Kern88]. */

for (i = 0, j = strlen(s) - 1; i < j; i++, j--)                    F 9
  c = s[i], s[i] = s[j], s[j] = c;
```

When the **SSOM** operator is applied on the body of the above loop, two mutants are generated. These are:

```
for (i = 0, j = strlen(s) - 1; i < j; i++, j--)

/* One left rotation generates this mutant. */                     M 19

  s[i] = s[j], s[j] = c, c = s[i];


for (i =0, j = strlen(s) -1; i < j; i++, j--)

/* Another left rotation generates this mutant. */                 M 20

  s[j] = c, c = s[i], s[i] = s[j];
```

When **SSOM** is applied on the `for` statement in the above program, it generates two more mutants, one by mutating the expression (i = 0, j=strlen(s)-1) to (j = strlen (s) - 1 , i = 0), and the other by mutating the expression (i++, j−−) to (j−−, i++). ■

[8]According to the syntax of C, each e_i can be an *assignment-expression*.

The **SSOM** operator is likely to generate several mutants that are equivalent to P. The mutants, generated by mutating the expressions in the `for` statement in the above example, are equivalent. In general, if the sub-expressions do not depend on each other then the mutants generated will be equivalent to P.

9.15 Move Brace Up or Down: SMVB

The closing brace (}) is used in C to indicate the end of a compound statement. It is possible for a programmer to incorrectly place the closing brace thereby including, or excluding, some statements within a compound statement. The **SMVB** operator models this error.

A statement immediately following the loop body is pushed inside the body. This corresponds to moving the closing brace *down* by one statement. The last statement inside the loop body is pushed out of the body. This corresponds to moving the closing brace *up* by one statement. A compound statement consisting of only one statement may not have explicit braces surrounding it. However, the beginning of a compound statement is considered to have an implied opening brace and the semicolon at its end is considered to be an implied closing brace[9].

Example 11

Consider the function in F 2. When mutated using the **SMVB** operator, it generates mutants M 21 and M 22. ■

```
/* This is a mutant generated by SMVB. In this one,
   the for loop body extends to include the s[n+1] = '\0' statement. */

int trim(char s[] )
{
  int n;
  for (n = strlen(s)-1; n >= 0; n--) {                                M 21
    if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
```

[9]To be precise, the semicolon at the end of the statement inside the loop body, is considered as a semicolon *followed by a closing brace.*

```
    break;
  s[n+1] = '\0';
  }
  return n;
}


/* This is another mutant generated by SMVB.
   In this one the for loop body becomes empty. */

int trim(char s[] )
{
  int n;
  for (n = strlen(s)-1; n >= 0; n--);                                    M 22
  if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
    break;
  s[n+1] = '\0';
  return n;
}
```

In certain cases, moving the brace may include, or exclude, a large piece of code. For example, suppose that a `while` loop with a substantial amount of code in its body, follows the closing brace. Moving the brace down will cause the entire `while` loop to be moved into the loop body that is being mutated. A C programmer is not likely to make such an error. However, there is a high probability of such a mutant being is killed.

9.16 Switch Statement Mutation: SSWM

Errors in the formulation of the cases in a `switch` statement are modeled by **SSWM**.

The expression e in the `switch` statement is replaced by the trap_on_case function. The input to this function is a condition formulated as $e = a$, where a is one of the case labels in the `switch` body. This generates a total of n mutants of a `switch` statement assuming that there are n case labels. In addition, one mutant is generated with the input condition for trap_on_case set to $e = d$, where d is computed as:

$$d = e! = c_1 \&\& e! = c_2 \&\& \ldots e! = c_n$$

The next example exhibits some mutants generated by **SSWM**.

Example 12

Consider the following program fragment:

```
/* This fragment is from a program on p59 of [Kern88].

switch(c) {
case '0': case '1': case '2': case '3':case '4': case '5':
case '6':case '7':case '8':case '9':
  ndigit[c-'0']++;
  break;
case ' ':
case '\n':
case '\t':                                                    F 10
  nwhite++;
  break;
default:
  nother++;
  break;
}
```

The **SSWM** operator will generate a total of 14 mutants for F 10. Two of them appear in M 23 and M 24. ■

```
switch(trap_on_case(c,'0')) {
case '0': case '1': case '2': case '3':case '4': case '5':
case '6':case '7':case '8':case '9':
  ndigit[c-'0']++;
  break;
case ' ':
case '\n':
case '\t':                                                                M 23
  nwhite++;
  break;
default:
  nother++;
  break
}


c'=c;  /* This is to ensure that side effects in c occur once. */
d = c' != '0' && c' != '1' && c' != '3' && c' != '4' && c' != '5' &&
  c' != '6' && c' != '7' && c' != '8' && c' != '9' && c' != '\n' && c' != '\t';
switch(trap_on_case(c', d)) {
⋮
/* switch body is the same as that in M 23. */                            M 24
⋮
}
```

A test set that kills all mutants generated by **SSWM** ensures that all cases, including the default case, have been covered. We refer to this coverage as *case coverage*. It is important to point out that the **STRP** operator may not provide case coverage especially when there is fall through code in the `switch` body. This also implies that some of the mutants generated when **STRP** mutates the cases in a `switch` body, may be equivalent to those generated by **SSWM**.

Example 13

Consider the following program fragment:

```
/* This is an example of fall thorough code. */
switch (c) {
case '\n':
  if (n == 1) {
    n−−;
    break;                                         F 11
  }
  putc('\n');
case '\r':
  putc('\r');
  break;
}
```

One of the mutants generated by **STRP** when applied on F 11 will have the putc('\r') in the second case replaced by trap_on_statement(). A test case that forces the expression c to evaluate to '\n' and n evaluate to any value not equal to 1, is sufficient to kill such a mutant. On the contrary, an **SSWM** mutant will encourage the tester to construct a test case that forces the value of c to be '\r'. ■

It may, however, be noted that both the **STRP** and the **SSWM** serve different purposes when applied on the `switch` statement. Whereas **SSWM** mutants are designed to provide case coverage, mutants generated when **STRP** is applied to a `switch` statement, are designed to provide statement coverage *within* the `switch` body.

Table 2: Classification of Binary Operators in C

Type	**Category**	**Operators**	**Code**¶
Non-assignment			
	Arithmetic	+ − * / %	**A**
	Bitwise	| & ˆ	**B**
	Logical	|| &&	**L**
	Shift	<< >>	**S**
	Relational	< > <= >= == !=	**R**
Assignment			
	Arithmetic	*= /= %= += −=	**A**
	Bitwise	&= ˆ= |=	**B**
	Plain	=	**E**
	Shift	<<= >>=	**S**

¶ Any ambiguities in the use of these codes are resolved from the context.

10 OPERATOR MUTATIONS

C provides a variety of operators for use in different contexts. For the purpose of defining mutant operators[10], the C operators are classified as shown in Table 2. This table lists sets of C operators used later as the domain and range of different mutant operators. The **Code** column in this table lists the single letter codes that are used in the name of a mutant operator. The context of their use enables the differentiation amongst categories with identical codes. Fig. 7 shows all the contexts.

In this section we describe the mutant operators that are designed to model errors in the use of C operators. These mutant operators are classified as shown in Fig. 6.

The design of operator level mutations was guided by the following facts:

- In C, operators belonging to one category can be used in a variety of contexts. For example, there is no entity such as an *arithmetic expression* within which only the arithmetic operators can be used. Thus, within an expression that computes arithmetic values, one could as well use logical operators. For example, for two integers a and b, both $a + b$ and $a \&\& b$ are valid

[10]In this section, we use the word *operator* to refer to the operators in C and also to mutant operators. Hence the prefix *C* or *mutant* is used to avoid any ambiguity.

Figure 6: Mutant operators for C operators.

expressions that compute integers[11].

- It is possible to group all the C operators in one set and define one mutant operator with respect to this set. There are two problems with this approach: (a) a large number of mutants will be generated whenever such a mutant operator is enabled and (b) the user will not have the flexibility of selectively mutating the C operators. Hence, this approach was rejected. We have classified the mutant operators so that the user has the maximum flexibility in their selection.

10.1 Binary Operator Mutations: Obom

The incorrect choice of a binary C-operator within an expression is the error modeled by this mutant operator.

Obom is a mutant operator category. Figure 7 shows the classification of **Obom**. As shown, **Obom** can be subdivided into mutant operators that belong to the two subcategories: *Comparable Operator Replacement* (**Ocor**) and *Incomparable Operator Replacement* (**Oior**). Within each subcategory, the mutant operators correspond to either the *non-assignment* or to the *assignment* operators in C. Each mutant operator, belonging to the **Obom** category, systematically *replaces* a C operator in its domain by operators in its range. The domain and range for all the mutant operators in the **Obom** category are specified in Tables 3 and 4. These tables also provide one example illustrating each mutant operator. In certain contexts, only a subset of arithmetic operators is used. For example, it is illegal to add two pointers, though a pointer may be subtracted from another. All mutant operators that mutate C-operators, are assumed to recognize such exceptional cases to retain the syntactic validity of the mutant.

10.2 Unary Operator Mutations: Ouor

Mutations in this subcategory consist of mutant operators that model errors in the use of unary operators and conditions. **Ouor** is further subdivided into five subcategories described below.

[11]Note that this is not true in some other languages such as Pascal.

Figure 7: Binary operator mutation operator.

Table 3: Domain and Range of Mutant Operators in **Ocor**

Name	**Domain**	**Range**	**Example**
OAAA	Arithmetic assignment	Arithmetic assignment	a+= b → a -= b
OAAN	Arithmetic	Arithmetic	a +b → a * b
OBBA	Bitwise assignment	Bitwise assignment	a &= b → a|= b
OBBN	Bitwise	Bitwise	a & b → a| b
OLLN	Logical	Logical	a && b → a ||b
ORRN	Relational	Relational	a < b → a <= b
OSSA	Shift assignment	Shift assignment	a <<= b → a >>= b
OSSN	Shift	Shift	a << b → a>> b

† Read X → Y as ‘X gets mutated to Y’.

Table 4: Domain and Range of Mutant Operators in **Oior**

Name	**Domain**	**Range**	**Example**
OABA	Arithmetic assignment	Bitwise assignment	a += b → a |= b
OAEA	Arithmetic assignment	Plain assignment	a += b → a = b
OABN	Arithmetic	Bitwise	a + b → a & b
OALN	Arithmetic	Logical	a + b → a && b
OARN	Arithmetic	Relational	a + b → a < b
OASA	Arithmetic assignment	Shift assignment	a += b → a <<= b
OASN	Arithmetic	Shift	a + b → a << b
OBAA	Bitwise assignment	Arithmetic assignment	a |= b → a += b
OBAN	Bitwise	Arithmetic	a & b → a + b
OBEA	Bitwise assignment	Plain assignment	a &= b → a =b
OBLN	Bitwise	Logical	a & b → a && b
OBRN	Bitwise	Relational	a & b → a < b
OBSA	Bitwise assignment	Shift assignment	a &= b → a <<= b
OBSN	Bitwise	Shift	a & b → a << b
OEAA	Plain assignment	Arithmetic assignment	a = b → a += b
OEBA	Plain assignment	Bitwise assignment	a =b → a &= b
OESA	Plain assignment	Shift assignment	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
OSAA	Shift assignment	Arithmetic assignment	a <<= b → a += b
OSAN	Shift	Arithmetic	a<< b → a + b
OSBA	Shift assignment	Bitwise assignment	a << b → a |= b
OSBN	Shift	Bitwise	a << b → a & b
OSEA	Shift assignment	Plain assignment	a <<= b → a = b
OSLN	Shift	Logical	a << b → a && b
OSRN	Shift	Relational	a << b → a < b

† Read X → Y as 'X gets mutated to Y'.

10.2.1 Increment/Decrement: Oido

The ++ and -- operators are used frequently in C programs. The **OPPO** and **OMMO** mutant operators model the errors that arise from the incorrect use of these C operators. The incorrect uses modeled are: (a) ++ (or --) used instead of -- (or ++) and (b) prefix increment (decrement) used instead of postfix increment (decrement).

The **OPPO** operator generates two mutants. An expression such as ++x is mutated to x++ and --x. An expression such as x++, will be mutated to ++x and x--. The **OMMO** operator behaves similarly. It mutates --x to x-- and ++x. It also mutates x-- to --x and x++. Both the operators will not mutate an expression if its value is not used. For example, an expression such as i++ in a `for` header will not be mutated, thereby avoiding the creation of an equivalent mutant. An expression such as *x++ will be mutated to *++x and *x--.

10.2.2 Logical Negation: OLNG

Often, the sense of the condition used in iterative and selective statements is reversed. **OLNG** models this error.

Consider the expression $x\ op\ y$, where op can be any one of the two logical operators: && and ||. **OLNG** will generate three mutants of such an expression as follows: $x\ op\ !\,y$, $!\,x\ op\ y$, and $!(x\ op\ y)$.

10.2.3 Logical context negation: OCNG

In selective and iterative statements, excluding the `switch`, often the sense of the controlling condition is reversed. **OCNG** models this error.
The controlling condition in the iterative and selection statements is negated. The following are illustrative examples:

`if` (expression) statement $\rightarrow$ `if` (!expression) statement

`if` (expression) statement `else` statement $\rightarrow$ `if` (!expression) statement `else` statement

`while` (expression) statement $\rightarrow$ `while` (!expression) statement

`do` statement `while` (expression) $\rightarrow$ `do` statement `while` (!expression) `for` (expression; expression; expression) statement $\rightarrow$ `for` (expression; !expression, expression) statement

expression ? expression : conditional expression $\rightarrow$!expression ? expression : conditional expression

OCNG may generate mutants with infinite loops when applied on an iteration statement. Further, it may also generate mutants generated by **OLNG**. Note that a condition such as (x<y) in an `if` statement will not be mutated by **OLNG**. However, the condition ((x<y&&p>q) will be mutated by both **OLNG** and **OCNG** to (!(x<y)&&(p>q)).

10.2.4 Bitwise Negation: OBNG

The sense of the bitwise expressions may often be reversed. Thus, instead of using (or not using) the one's complement operator, the programmer may not use (or may use) the bitwise negation operator. The **OBNG** operator models this error.

Consider an expression of the form $x\ op\ y$, where op is one of the bitwise operators: | and &. The **OBNG** operator mutates this expression to: $x\ op\,\tilde{}\ y$, $\tilde{}x\ op\ y$, and $\tilde{}(x\ op\ y)$. **OBNG** does not consider the iterative and conditional operators as special cases. Thus, for example, a statement such as `if` (x && a | b) p = q will get mutated to the following statements by **OBNG**:

`if` (x && a |˜b) p = q

`if` (x &&˜a | b) p = q

`if` (x &&˜(a | b)) p = q

10.2.5 Indirection Operator Precedence Mutation: OIPM

Expressions constructed using a combination of ++, --, and the indirection operator (*), can often contain precedence errors. For example, using *p++ when (*p)++ was meant, is one such error. **OIPM** operator models such errors.

OIPM mutates a reference of the form $*\,x\ op$ to $(*\,x)\ op$ and $op\ (*\,x)$, where op can be ++ and --. Recall that in C, $*\,x\ op$ implies $*\,(x\ op)$. If op is of the form [y], then only $(*\,x)\ op$ is generated. For example, a reference such as $*x[p]$ will be mutated to (*x)[p].

The above definition is for the case when only one indirection operator has been used to form the reference. In general, there could be several indirection operators used in formulating a reference. For example, if x is declared as `int` $***x$, then $***$x++ is a valid reference in C. A more general definition of **OIPM** takes care of this case.

Consider the following reference:

$$\underbrace{*\ *\ldots*}_{n}\ x\ op$$

OIPM will systematically mutate the above reference to the following references:

$$\begin{array}{ccc}
\underbrace{*\ *\ldots*}_{n-1} & (*\,x) & op \\
\underbrace{*\ *\ldots*}_{n-1} & op & (*\,x) \\
\underbrace{*\ *\ldots*}_{n-2} & (*\,*\ x) & op \\
\underbrace{*\ *\ldots*}_{n-2} & op & (*\,*\ x) \\
\vdots & \vdots & \vdots \\
* & (\underbrace{*\ *\ldots*}_{n-1}\ x) & op \\
* & op & (\underbrace{*\ *\ldots*}_{n-1}\ x) \\
 & (\underbrace{*\ *\ldots*}_{n}\ x) & op \\
 & op & (\underbrace{*\ *\ldots*}_{n}\ x)
\end{array}$$

Multiple indirection operators are used infrequently. Hence, in most cases, we expect **OIPM** to generate two mutants for each reference involving the indirection operator.

10.2.6 Cast operator replacement: OCOR

A cast operator, referred to as *cast*, is used to explicitly indicate the type of an operand. Errors in such usage are modeled by **OCOR**.

Every occurrence of a cast operator is mutated by **OCOR**. Casts are mutated in accordance with the restrictions listed below. These restrictions are derived from the rules of C as specified in [Kern88]. While reading the cast mutations described below, $\leftrightarrow$ may be read as 'gets mutated to'. All entities to the left of $\leftrightarrow$ get mutated to the entities on its right, and vice-versa. The notation X* can be read as 'X and all mutations of X excluding duplicates'.

`char`	$\leftrightarrow$	`signed char`	`unsigned char`
		`int`*	`float`*
`int`	$\leftrightarrow$	`signed int`	`unsigned int`
		`short int`	`long int`
		`signed long int`	`signed long int`
		`float`*	`char`*
`float`	$\leftrightarrow$	`double`	`long double`
		`int`*	`char`*
`double`	$\leftrightarrow$	`char`*	`int`*
		`float`*	

Example 14

Consider the statement:

`return (unsigned int)` (next/65536) % 32768

Sample mutants generated when **OCOR** is applied to the above statement are shown below. ■

```
short int  long int
float  double
```

Note that cast operators other than those described in this section, are not mutated. For example, the casts in the following statement are not mutated:

qsort((`void` **) lineptr, 0, nlines-1, (`int`(*) (`void`*, `void`*))(numeric ? numcmp :strcmp))

Figure 8: Classification of variable mutations.

The decision not to mutate certain casts was motivated by their infrequent use and the low probability of an error that could be modeled by mutation. For example, a cast such as `void` ** is not used very often and when it is used the chances of it being mistaken for, say, an `int`, are low.

11 VARIABLE MUTATIONS

Incorrect use of identifiers can often induce program errors that remain unnoticed for quite long. Variable mutations are designed to model such errors. Fig. 8 shows the subcategories of mutant operators in the variable mutations category. The classification ensures that syntactically correct mutants are generated. It also enables the tester to control the generation of mutants. Table 5 lists the domains of all variable mutation operators.

Table 5: Domain of Mutant Operators in Variable Mutations Category

Operator or subcategory	**Meaning**	**Domain**
Varr	Mutate array references	Arrays in expressions
Vprr	Mutate pointer references	Pointers in expressions
Vsrr	Subcategory of mutant operators to mutate scalars	Scalar variables in an expression
Vtrr	Mutate structure references	Structures in expressions
VASM	Mutate subscripts in array references	Array subscripts
VSCR	Mutate components of a structure	Structure components within expression

11.1 Scalar Variable Reference Replacement: Vsrr

Use of a wrong scalar variable is the error that is modeled by **Vsrr**.

Vsrr is a set of two mutant operators: **VGSR** and **VLSR**. **VGSR** mutates all scalar variable references by using GS'_f as the range. **VLSR** mutates all scalar variable references by using LS'_f as the range of the mutant operator. Types are ignored during scalar variable replacement. For example, if i is an integer and x a real, i will be replaced by x and vice versa.

Entire scalar references are mutated. For example, if *screen* is as declared above, and $screen.p1.x$ is a reference, then the entire reference, i.e. $screen.p1.x$ will be mutated. $p1$ or x, will not be mutated separately by any one of these two operators. The individual components of a structure may be mutated by the **VSCR** operator. *screen* itself *may*[12] be mutated by one of the **Vtrr** operators. Similarly, in a reference such as $*p$, for p as declared above, $*p$ will be mutated. p alone *may* be mutated by one of the **Vprr** operators. As another example, the entire reference $q\,[\,i\,]$ will be mutated, q itself *may* be mutated by one of the **Varr** operators.

11.2 Array Reference Replacement: Varr

Incorrect use of an array variable is the error modeled by **Varr**.

[12] We often say that an entity x *may* be mutated by an operator. This implies that there may be no other entity y to which x can be mutated.

Varr is a set of two mutant operators: **VGAR** and **VLAR**. These operators mutate an array reference in function f using, respectively, the sets GA'_f and LA'_f. Types[13] are preserved while mutating array references. Thus, if a and b are, respectively, arrays of integers and pointers to integers, a will not be replaced by b and vice-versa.

11.3 Structure Reference Replacement: Vtrr

Incorrect use of a variable of type structure, is the error modeled by **Vtrr**.

Vtrr is a set of two mutant operators: **VGTR** and **VLTR**. These operators mutate a structure reference in function f using, respectively, the sets GT'_f and LT'_f. Types are preserved while mutating structures. For example, if s and t denote two structures of different types[14], then s will not be replaced by t and vice-versa.

11.4 Pointer Reference Replacement: Vprr

Incorrect use of a pointer variable is the error modeled by **Vprr**.

Vprr is a set of two mutant operators: **VGPR** and **VLPR**. These operators mutate a pointer reference in function f using, respectively, the sets GP'_f and LP'_f. Types are preserved while performing mutation. For example, if p and q are pointers to an integer and structure, respectively, then p will not be replaced by q, or vice-versa.

11.5 Structure Component Replacement: VSCR

Often one may use the wrong component of a structure. **VSCR** models such errors[15].

Let s be a variable of some structure type. Let $s.c_1.c_2 \ldots .c_n$ be a reference to one of its components declared at *level* n within the structure. c_i, $1 \leq i \leq n$ denotes an identifier declared at level i within s. **VSCR** systematically mutates each identifier at level i, by all the other type compatible identifiers at the same level.

As an example, consider the following structure:

[13] We assume name equivalence of types as defined in C.

[14] We assume *name* equivalence for types as in C.

[15] *Structure* refers to data elements declared using the `struct` type specifier.

```
struct example {
  int x;
  int y;
  char c;                                                    F 12
  int d[10];
}

struct example s, r;
```

The reference $s.x$ will be mutated to $s.y$ and $s.c$ by **VSCR**. Another reference $s.d[j]$ will be mutated to $s.x$, $s.y$, and $s.c$. Note that the reference to s itself will be mutated to r by one of the **Vsrr** operators.

Next, suppose that we have a pointer to *example* declared as:

```
struct example *p;
```

A reference such as $p->x$ will be mutated to $p->y$ and $p->c$.

Now, consider the following recursive structure:

```
struct tnode {
  char *word;
  int count;
  struct tnode *left;
  struct tnode *right;                                       F 13
}

struct tnode *q;
```

A reference such as $q->left$ will be mutated to $q->right$. Note that $left$, or any field of a structure, will *not* be mutated by any of the **Vsrr** operators. This is because a field of a structure does not belong to any of the global or local sets, or reference sets, defined earlier. Also, a reference such as $q\!-\!>\!count$ will not be mutated by **VSCR** because there is no other compatible field in F13.

11.6 Array reference subscript mutation: VASM

While referencing an element of a multidimensional array, the order of the subscripts may be incorrectly specified. **VASM** models this error.

Let a denote an n-dimensional array, $n > 1$. A reference such as:

$$a\,[e_1][e_2]\ldots[e_n]$$

with e_i, $1 \le i \le n$ denoting a subscript expression, will be mutated by *rotating* the subscript list. Thus, the above reference generates the following $(n-1)$ mutants when **VASM** is applied:

$$\begin{aligned} a\,[e_n]\;[e_1]\ldots[e_{n-2}]\;[e_{n-1}] \\ a\,[e_{n-1}]\;[e_n]\ldots[e_{n-3}]\;[e_{n-2}] \\ \vdots \\ a\,[e_2]\;[e_3]\ldots[e_n]\;[e_1] \end{aligned}$$

11.7 Domain Traps: VDTR

Statements containing scalar references are affected. **VDTR** provides domain coverage for scalar variables. The domain is partitioned into three subdomains: negative values, zero, and positive values.

VDTR mutates each scalar reference x of type t in an expression, by $f(x)$, where f could be one of the several functions shown in Table 6. Note that all functions listed in Table 6 for a type t are applied on x. When any of these functions is executed, the mutant is killed. Thus, if i, j, and k are pointers to integers, then the statement:

$$*i = *j + *k++$$

will be mutated by **VDTR** to:

Table 6: Functions Used by the **VDTR** Operator

Function[†] introduced	**Description**
trap_on_negative_x	Mutant killed if argument is negative, else return argument value.
trap_on_positive_x	Mutant killed if argument is positive, else return argument value.
trap_on_zero_x	Mutant killed if argument is zero, else return argument value.

[†] x can be integer, real, or double. It is integer if the argument type is `int`, `short`, `signed`, or `char`. It is real if the argument type is `float`. It is double if the argument is of type `double` or `long`.

$$
\begin{aligned}
*i &= \text{trap\_on\_zero\_integer}(*j) + *k++ \\
*i &= \text{trap\_on\_positive\_integer}(*j) + *k++ \\
*i &= \text{trap\_on\_negative\_integer}(*j) + *k++ \\
*i &= *j + \text{trap\_on\_zero\_integer}(*k++) \\
*i &= *j + \text{trap\_on\_positive\_integer}(*k++) \\
*i &= *j + \text{trap\_on\_negative\_integer}(*k++) \\
*i &= \text{trap\_on\_zero\_integer}(*j + *k++) \\
*i &= \text{trap\_on\_positive\_integer}(*j + *k++) \\
*i &= \text{trap\_on\_negative\_integer}(*j + *k++)
\end{aligned}
$$

In the above example, $*k++$ is a reference to a scalar, therefore the trap function has been applied to the entire reference. Instead, if the reference was (*k)++, then the mutation would be $f(*k)++$, f being any of the relevant functions.

11.8 Twiddle Mutations: VTWD

Values of variables or expressions can often be off the desired value by ± 1. The twiddle mutations model such errors. Twiddle mutations are useful for checking boundary conditions for scalar variables.

Each scalar reference x is replaced by $pred(x)$ and $succ(x)$, where $pred$ and $succ$ return, respectively, the immediate predecessor and the immediate successor of the current value of the argument. When applied on a real argument, a small value is added (by $succ$) to or subtracted (by $pred$) from the argument. This value can be user defined, such as $\pm.01$, or may default to an implementation defined value.

Example 15

Consider the assignment:

$$p = a + b$$

Assuming that p, a, and b are integers, **VTWD** will generate the two mutants shown below. ■

$$\begin{aligned} p &= a + b + 1 \\ p &= a + b - 1 \end{aligned}$$

Pointer variables are not mutated. However, a scalar reference constructed using a pointer is mutated as defined above. For example, if p is a pointer to an integer, then $*p$ will be mutated. Some mutants may cause overflow or underflow errors implying that they are killed.

12 CONSTANT MUTATIONS

In this section we define mutant operators related to the use of constants. These operators model coincidental correctness and, in this sense, are similar to scalar variable replacement operators.

12.1 Required Constant Replacement: CRCR

Let I and R denote, respectively, the sets $\{0, 1, -1, u_i\}$ and $\{0.0, 1.0, -1.0, u_r\}$. u_i and u_r denote user specified integer and real constants, respectively. Use of a variable where an element of I or R was the correct choice, is the error modeled by **CRCR**.

Each scalar reference is replaced systematically by elements of I or R. If the scalar reference is integral, I is used. For references that are of type floating[16], R is used. Reference to an entity via a pointer is replaced by `null`. Left operands of assignment operators, $++$, and $--$ are not mutated.

Example 16

Consider the statement:

$$k = j + *p$$

where k and j are integers and p is a pointer to an integer. When applied on the above statement, **CRCR** will generate the mutants given below. ■

$$\begin{aligned} \mathrm{k} &= 0 + {}^*\mathrm{p} \\ \mathrm{k} &= 1 + {}^*\mathrm{p} \\ \mathrm{k} &= 1 + {}^*\mathrm{p} && \text{M 25} \\ \mathrm{k} &= \mathrm{u}_i + {}^*\mathrm{p} \\ \mathrm{k} &= \mathrm{j} + \texttt{null} \end{aligned}$$

A **CRCR** mutant encourages a tester to design at least one test case that forces the variable replaced to take on values other than from the set I or R. Thus, such a mutant attempts to overcome coincidental correctness of P.

12.2 Constant for Constant Replacement: Cccr

Just as a programmer mistakenly use one identifier for another, a possibility exists that one may use one constant for another. Mutant operators in the **Cccr** category model such errors.

[16]The terms *integral* and *floating* have the same meaning as in [Kern88].

Cccr is a set of two mutant operators: **CGCR** and **CLCR**. **CGCR** and **CLCR** mutate constants in f using, respectively, the sets GC_f and LC_f.

Example 17

Suppose that a constant 5 appears in an expression, and $GC_f = \{0, 1.99, 'c'\}$, then 5 will be mutated to 0, 1.99, and $'c'$ thereby producing three mutants. ■

Pointer constant, `null`, is not mutated. Left operands of assignment, ++, and −− operators are also not mutated.

12.3 Constant for Scalar Replacement: Ccsr

Use of a scalar variable, instead of a constant, is the error modeled by mutant operators in the **Ccsr** category.

Ccsr is a set of two mutant operators: **CGSR** and **CLSR**. **CGSR** mutates all occurrences of scalar variables or scalar references by constants from the set GC_f. **CLSR** is similar to **CGSR** except that it uses LC_f for mutation. Left operands of assignment, ++, and −− operators are not mutated.

13 COMPARISON OF MUTANT OPERATORS FOR C AND FORTRAN 77

C has a total of 77 mutant operators as compared to only 22 for Fortran 77. The following differences between the two languages have been largely responsible for the complexity of the mutant operator set of C:

1. C has more primitive types than Fortran 77. Further, types in C can be mixed in a variety of combinations. This has resulted in a large number of **Obom** operators.
2. The statement structure of C is recursive. Fortran 77 has *single line* statements. Though this has not resulted in more operators being defined for C it has, however, made the definition of operators such as **SSDL** and **SRSR** significantly different from the definition of the corresponding operators in Fortran 77.

3. Scalar references in C can be constructed non-trivially using functions, pointers, structures, and arrays. In Fortran 77, only functions and arrays can be used to construct scalar references. This has resulted in operators like the **VSCR** and several others in the **Vsrr** category. Note that in Fortran 77, **SVR** is one operator whereas in C, it is a set of several operators.

4. C has a *comma* operator not in Fortran 77. This has resulted in the **SSOM** operator.

5. All iterations, or the current iteration, of a loop can be terminated in C using, respectively, the `break` and `continue` statements. Fortran 77 provides no such facility. This has resulted in additional mutant operators such as **SBRC**, **SCRB**, and **SBRn**.

Table 7 lists all the Fortran 77 mutant operators and the corresponding semantically nearest C mutant operator

14 FUTURE WORK

Currently a system for mutating C programs is under development. When completed, this system will be a part of the software testing environment. To speed up the testing process, we have resorted to object level mutation and parallel computing. Object level mutation lets the compiler apply mutation operators so that the object code of the mutants is obtained which can then be executed by the hardware. Parallel machines, such as the Alliant FX/80 and the Ncube/7, are being to used for executing a large number of mutants in parallel. The design of an advanced debugger is also underway. This debugger is expected to help the tester in program debugging in co-operation with the mutation based test tool. We expect that these approaches will help make mutation based software testing for C programs a practically viable technique.

ACKNOWLEDGEMENTS

We would like to thank Elizabeth Northern for all the help she rendered for the preparation of this report.

Table 7: A Comparison of Fortran 77 and C Mutant Operators

Fortran 77 operator	**Description**	**Semantically nearest C operator/category**
AAR	Array reference for array reference	**Vsrr**
ABS	Absolute value insertion	**VDTR**
ACR	Array reference for constant replacement	**Vsrr**
AOR	Arithmetic operator replacement	**OAAN**
ASR	Array reference for scalar variable replacement	**Vsrr**
CAR	Constant for array reference replacement	**Ccsr**
CNR	Comparable array name replacement	**Vsrr**
CRP	Constant replacement	**CRCR**
CSR	Constant for scalar replacement	**Ccsr**
DER	DO statement *END* replacement	**OTT**
DSA	DATA statement alterations	None
GLR	GOTO label replacement	**SGLR**
LCR	Logical connector replacement	**OBBN**
ROR	Relational operator replacement	**ORRN**
RSR	Return statement replacement	**SRSR**
SAN	Statement Analysis	**STRP**
SAR	Scalar variable for array reference replacement	**Vsrr**
SCR	Scalar for constant replacement	**Vsrr**
SDL	Statment deletion	**SSDL**
SRC	Source constant replacement	**CRCR**
SVR	Scalar variable replacement	**Vsrr**
UOI	Unary operator insertion	**OLNG**, **VTWD**

References

[DeMi79] A.T. Acree, R.A. DeMillo, T.A. Budd and F.G. Sayward, "Mutation Analysis," *Technical Report*, GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA 30332, 1979.

[Basi84] V.R. Basili and B.T. Perricone, "Software errors and complexity," *Comm. ACM*, vol. 27, no. 1, pp. 42-52, Jan. 1984.

[Budd78] T. A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "The design of a prototype mutation system for program testing," *Proceedings NCC 1978*, 1978, Alexandria, Va, 1978.

[Budd80] T. A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*,Las Vegas, 1980.

[DeMi88] B.J. Choi, R. A. DeMillo, E.W. Krauser, A.P. Mathur, R.J. Martin, A.J. Ofutt, H. Pan, and E.H. Spafford, "The Mothra Toolset," *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, January 3-6, 1989.

[DeMi87] R.A. DeMillo, D.S. Guindi, K.N. King and W.M. McCracken, "An Extended Overview of the Mothra Testing Environment," *Proceedings Second Workshop On Software Testing, Verification, and Analysis*, 19-21 July, 1988, Banff, Canada.

[DeMM87] R.A. DeMillo, W. Michael McCracken, R.J. Martin, and John F. Passafume, "Software Testing and Evaluation," *The Benjamin/Cummings Publishing Company, Inc.* Menlo park, CA, 1987.

[Howd87] William E. Howden, "Functional Program Testing and Analysis,", *McGraw Hill Book Company, New York*, 1987.

[Kern88] Brian Kernighan and Dennis M. Ritchie, "The C Programming Language," *Prentice Hall, New Jersey*, Second Edition, 1988.

APPENDIX A: INDEX TO MUTANT OPERATORS/CATEGORIES

List of Mutant Operator Categories for ANSI C

Operator	**Description**	**Page**§
Oarr	Array reference replacement	56
Obor	Binary operator replacement	48
Cccr	Constant for constant replacement	62
Ocor	Comparable operator replacement	48
Ccsr	Constant for scalar replacement	63
Oior	Incomparable operator replacement	48
Oido	Increment/decrement	51
Vprr	Pointer reference replacement	57
Vsrr	Scalar variable replacement	56
Vtrr	Structure reference replacement	57

§ All page numbers in this Appendix refer to the page of first definition of the mutant operator/category.

List of Mutant Operators for ANSI C

Operator	Domain	Description	Page
CGCR	Constants	Constant replacement using global constants	63
CLSR	Constants	Constant for scalar replacement using local constants	63
CGSR	Constants	Constant for scalar replacement using global constants	63
CRCR	Constants	Required constant replacement	62
CLCR	Constants	Constant replacement using local constants	63
OAAA	‡	arithmetic assignment mutation	49
OAAN	‡	arithmetic operator mutation	49
OABA	†	arithmetic assignment by bitwise assignment	50
OABN	†	arithmetic operator by bitwise operator	50
OAEA	†	arithmetic assignment by plain assignment	50
OALN	†	arithmetic operator by logical operator	50
OARN	†	arithmetic operator by relational operator	50
OASA	†	arithmetic assignment by shift assignment	50
OASN	†	Arithmetic operator by shift operator	50
OBAA	†	Bitwise assignment by arithmetic assignment	50
OBAN	†	Bitwise operator by arithmetic assignment	50
OBBA	‡	Bitwise assignment mutation	49
OBBN	‡	Bitwise operator mutation	49
OBEA	†	Bitwise assignment by plain assignment	50
OBLN	†	Bitwise operator by logical operator	50
OBNG	†	Bitwise negation	52
OBRN	†	Bitwise operator by relational operator	50
OBSA	†	Bitwise assignment by shift assignment	50
OBSN	†	Bitwise operator by shift operator	50
OCOR	Casts	Cast operator by cast operator	53
OEAA	†	Plain assignment by arithmetic assignment	50
OEBA	†	Plain assignment by bitwise assignment	50
OESA	†	Plain assignment by shift assignment	50

† See Table 4.

‡ See Table 3.

List of Mutant Operators for ANSI C (*contd.*)

Operator	Domain	Description	Page
OIPM	Expresions	Indirection operator precedence mutation	52
OLAN	†	Logical operator by arithmetic operator	50
OLBN	†	Logocal operator by bitwise operator	50
OLLN	‡	Logical operator mutation	49
OLNG	†	Logical negation	51
OLRN	†	Logical operator by relational operator	50
OLSN	†	Logical operator by shift operator	50
ORAN	†	Relational operator by arithmetic operator	50
ORBN	†	Relational operator by bitwise operator	50
ORLN	†	Relational operator by Logical operator	50
ORRN	‡	Relational operator mutation	49
ORSN	†	Relational operator by shift operator	50
OSAA	†	Shift assignment by arithmetic assignment	50
OSAN	†	Shift operator by arithmetic operator	50
OSBA	†	Shift assignment by bitwise assignment	50
OSBN	†	Shift operator by bitwise operator	50
OSEA	†	Shift assignment by plain assignment	50
OSLN	†	Shift operator by logical operator	50
OSRN	†	Shift operator by relational operator	50
OSSA	‡	Shift operator mutation	49
OSSN	‡	Shift assignment mutation	49
SBRC	`break`	`break` replacement by `continue`	34
SBRn	`break`	Break out to n^{th} level	35
SCRB	`continue`	`continue` replacement by `break`	34
SDWD	`do-while`	`do-while` replacement by `while`	36
SGLR	`goto`	`goto` label replacement	34
SMVB	Statement	Move brace up and down	41
SRSR	`return`	`return` replacement	32
SSDL	Statement	Statement deletion	30
SSOM	Statement	Sequence Operator Mutation	39
STRI	`if` Statement	Trap on `if` condition	29
STRP	Statement	Trap on statement execution	28

† See Table 4.

‡ See Table 3.

List of Mutant Operators for ANSI C (*contd.*)

Operator	Domain	Description	Page
SMTC	Iterative statements	n-trip continue	38
SSWM	`switch` statement	`switch` statement mutation	42
SMTT	Iterative statement	n-¡trip trap	37
SWDD	`while`	`while` replacement by `do-while`	36
VASM	Array subscript	Array reference subscript mutation	59
VDTR	Scalar reference	Absolute value mutation	59
VGAR	Array reference	Mutate array references using global array references	57
VGLA	Array reference	Mutate array references using both global and local array references	
VGPR	Pointer reference	Mutate pointer references using global pointer references	57
VGSR	Scalar reference	Mutate scalar references using global scalar references	56
VGTR	Structure reference	Mutate structure references using global structure references	57
VLAR	Array reference	Mutate array references using local array references	57
VLPR	Pointer reference	Mutate pointer references using local pointer references	57
VLSR	Scalar reference	Mutate scalar references using local scalar references	56
VLTR	Structure reference	Mutate structure references using only local structure references	57
VSCR	Strcuture component	Structure component replacement	57
VTWD	Scalar expression	Twiddle mutations	61

† See Table 4.

‡ See Table 3.

APPENDIX C: REVISION HISTORY

Version	Revision	Date	Remarks
1	0	November 27, 1988	Original draft
1	1	February 9, 1989	
1	2	October 3, 1994	