Constrained Mutation in C programs *

W. Eric Wong, José C. Maldonado, Marcio E. Delamaro, and Aditya P. Mathur
{ewong@hns.com, {jcmaldon, med}@icmsc.sc.usp.br, apm@cs.purdue.edu}

Abstract

Software development is always under the pressure of time and budget constraints before release. A good testing strategy should not only be effective and economical but also incremental. Although mutation testing has been empirically found to be effective in detecting faults, it remains unused for reasons of economics. A major obstacle to the use of mutation testing is its high computational cost. In this paper we report results from experiments designed to investigate six different constrained mutation mechanisms. Our data indicate that these alternatives not only reduce the cost of mutation significantly in terms of the number of test cases required and the number of mutants to be examined, but also maintain very good fault detection effectiveness. Effects of incremental mutation testing examining different sets of mutants are also discussed. Furthermore, our experiments are unique in that constrained mutation was performed directly on C programs. This eliminates the possible bias experienced by earlier mutation studies because of the programming language translation between the Fortran, Pascal, and C.

Keywords: Mutation, constrained mutation, fault detection effectiveness

1 Introduction

Software testing involves a cycle of generating test cases, observing program behavior, determining a failure, debugging the program, and removing faults. This process is always under the pressure of time and budget constraints. Not only must software be released to meet the market demand, but it also has to be properly tested to assure its quality. For a given software P, the following questions have to be answered: How is a test set T for P generated? How good is T in detecting faults in P? Should testing on P be stopped after applying T? and How can T be improved?

A good testing strategy should not only be *effective* and *economical* but also *incremental*. In other words, it should have a strong capability to detect faults, not be too expensive to use, and it should provide a flexible mechanism to "work around" under different constraints. If the relative cost-benefit of satisfying criteria C_1 and C_2 is known, then one may begin by using a criterion with a lower ratio of cost to benefit. Project economics permitting, after the criterion with the best cost-benefit has been satisfied, one may decide to use other criteria.

Although mutation testing has been empirically found to be very effective among the many testing techniques studied in the past decade in detecting faults [5, 10, 21], it remains unused for

* W. Eric Wong is with the Department of Software Technology, Hughes Network Systems, Germantown, MD 20876, USA. José C. Maldonado and Marcio E. Delamaro are with Software Engineering Laboratory, Department of Computer Science and Statistics, University of São Paulo, 13560-970, São Carlos, Brazil. Aditya P. Mathur is with the Department of Computer Science, Purdue University, W. Lafayette, IN 47907, USA.

economic reasons. The major obstacle to the use of mutation testing is its high computational cost. Since a significant portion of this cost is incurred in generating, compiling and executing mutants against one or more test cases, this cost can be lowered significantly by reducing the number of mutants considered while testing a program. In addition to reducing the computational cost, this approach also leads to fewer mutants to be manually examined by testers to determine equivalence. This improvement appears to be more significant when the software under test becomes more and more complicated. One way to reduce the number of mutants is to use *constrained mutation* [15, 22]. In constrained mutation the mutant generation process is constrained by fixing the mutant types to be generated. Only mutants of selected types are generated and examined. All other mutants are ignored.

On the other hand, different algorithms were proposed to reduce the time to execute *all* mutants on a parallel [12] or a sequential [18] machine. These algorithms either suffer a conversion between different operating environments or extra memory requirement to store the program state, which limits their usability. In either of the above cases, the number of mutants generated remains the same; it is the time to execute the mutants that is targeted for reduction. Though any reduction in testing time is welcome, it does not reduce the number of mutants that need to be identified as equivalent. Therefore, neither of them provides any improvement in reducing human effort in identifying equivalent mutants. In summary, these algorithms do not reduce the number of mutants to be examined or identified as equivalent; they only reduce the mutant execution time, whereas all three of them are reduced in our approach.

In this paper, we examine six different constrained mutation criteria. Details of these criteria are discussed in Section 2.3. For these criteria, we ask the following questions: What are their costs and and how effective is their fault detection? What is the additional cost-benefit of using another constrained mutation with more mutants to examine after a constrained mutation with fewer mutants to examine has been satisfied? Cost is measured in terms of the number of test cases required to satisfy a criterion and the number of mutants that need to be examined. Fault detection effectiveness is measured in terms of the percentage of adequate test sets with respect to a given criterion which detects at least one fault in the software under test. One unique characteristic of our experiments is that the constrained mutation was performed directly on C programs. This eliminates the possible bias experienced by earlier mutation studies because of the programming language translation between the Fortran, Pascal, and C.

The remainder of this paper is organized as follows. Section 2 provides an overview of mutation based adequacy criteria. Section 3 explains how we compared the cost and fault detection effectiveness of various test adequacy criteria. The experimental methodology is described in Section 4. Data collected from experiments and its analysis appear in Section 5. Our conclusions appear in Section 6.

2 An overview of mutation based testing

Details of mutation based testing can be found in [10, 14]. Below we present only the information required for an understanding of the remaining sections. Let P denote a program under test. D

is the set of all possible test cases in the input domain of P. A test case t is a sequence of inputs that is input to P during one execution of P. T is a set of one or more test cases on which P is executed during testing.

Let m be a syntactically correct program obtained by making a syntactic change in P; m is known as a *mutant* of P. Let r be a rule according to which P is changed; r is also known as a *mutant operator*. There could potentially be an infinite number of mutant operators. However, to keep mutation testing within reasonable bounds, the set of mutant operators is kept small and consists of simple mutant operators [6, 8]. Consider, for example, a mutant operator that generates two mutants of P by replacing a use of x by $x + 1$ and $x - 1$. When applied to a program containing the assignment statement $z := x + y$, this mutant operator will generate two mutants of P, one obtained by replacing this assignment by $z := x + 1 + y$ and the other by replacing this assignment by $z := x - 1 + y$.

The application of a set of mutant operators R to P results in a set of mutants $m_1, m_2, \ldots, m_n$, $n \geq 0$. Mutant m_i is considered *equivalent* to P if for all $t \in D$, $P(t) = m_i(t)$. When executed against a test case t, mutant m_i is considered *distinguished* from P if $P(t) \neq m_i(t)$. Unless distinguished, a non-equivalent mutant is considered *live*. One can obtain a variety of mutation based criteria by varying R. A test set T may be evaluated against the mutation criterion by executing each mutant against elements of T. The ratio of the number of mutants distinguished to the number of non-equivalent mutants is the *mutation score* of T for P. T is considered adequate with respect to a mutation criterion if the mutation score is the unity.

2.1 Random sampling mechanism

Several ways have been proposed to reduce the number of mutants to consider while testing a program. Acree [1], Sayward and Budd [4] examined the idea of using only a small randomly selected subset of all possible mutants generated in `Mothra` [6, 9]. Both of their experiments showed that a test set that is adequate with respect to randomly selected 10% of all mutants in `Mothra` is over 99% adequate with respect to these mutants. However, selecting mutants randomly ignores the fault detection capability of individual mutant types. Budd's fault detection experiments found that mutants generated with respect to one mutant operator may be more effective in detecting certain types of faults than mutants generated with respect to another operator. This suggests that while mutants are selected for examination, they should be weighted differently depending on their respective fault detection capability.

2.2 Comparisons of two selection mechanisms used in constrained mutation

A variant of Budd's and Acree's idea was proposed and termed *constrained mutation* [13]. In constrained mutation only mutants of selected types are generated and examined. All other mutants are ignored. One advantage of this approach is that it does not require the generation of all mutants prior to sampling. Based on this mechanism, Wong and Mathur [15, 21, 22] conducted experiments to investigate the cost-benefit of the *abs/ror* constrained mutation[1].

[1]The *abs* mutant operator generates mutants by replacing each use of x by $abs(x)$, $-abs(x)$, and $zpush(x)$ wherever possible. The *ror* mutant operator generates mutants by replacing each relational operator by other

Their data show that use of *abs/ror* mutation, compared to *mutation*[2], leads to at least an 80% reduction in the number of mutants to be examined and a 40% to 58% reduction in the number of test cases needed for adequacy. Such gain is accompanied by a loss of less than 5% in the ability to distinguish non-equivalent mutants and cover feasible all-uses.

This idea of the constrained mutation discussed above, renamed as *selective mutation*, was also investigated by Offutt [17]. In his experiments, mutant operators such as the one which generates mutants by replacing one scalar variable by another scalar variable were excluded because these operators are responsible for a large percentage of a program's mutants. The major difference between Wong and Mathur's constrained mutation and Offutt's mechanism is that they select mutant operators based on their significance in fault detection, whereas Offutt excludes mutant operators based on their number of mutants. Offutt's results show that at most a 60% reduction in the number of mutants to be examined leads to a 99% mutation score. No data on the loss of all-uses coverage were reported by him.

2.3 Mutant operators selected in our experiments

One significant issue that arises while using constrained mutation is how to select an effective small set of mutant operators. A good selection should not only reduce the execution cost dramatically but also maintain very good fault detection effectiveness. Among the seventy-one mutant operators used in PROTEUM [8], eleven of them were selected in our experiments. The rationale for selecting these mutant operators is enumerated below. Examples of these mutants appear in the appendix.

- *vdtr* and *vtwd* mutants:

  To distinguish *vdtr* mutants, a *positive*, a *negative*, and a *zero* value are necessary for the mutated expression when execution reaches that point. This requirement forces a tester to select test cases from different parts of the input domain. Similarly, distinguishing *vtwd* mutants forces a tester to select test cases from nearby parts of the input domain of each scalar reference. Test cases so generated may possibly make the program fail if faults exist. Note that Acree [1] also investigated a similar mutant operator like *vdtr*.

- *orrn*, *olln*, *olng*, and *ocng* mutants:

  When these mutant operators are applied to a program, mutants are generated at all points where a relational or a logical operator can be replaced by its alternatives or its negation, respectively. Distinguishing these mutants forces a tester to construct test cases which examine points on or near a predicate border. Such a process has been shown [3, 4, 7] to be effective in exposing certain types of faults such as the domain faults defined by Howden [11].

---

relational operators. In addition, it also replaces each condition consisting of at least one relational operator by the boolean constants `true` and `false`.

[2]The mutation criterion referred here is defined with respect to the set of 22 mutant operators used in Mothra.

- *orln*, *olrn*, *olan*, and oaln mutants:

  When these mutant operators are applied to a program, mutants are generated at all points where a logical operator can be replaced by a relational or an arithmetic operator and vice versa. Distinguishing these mutants helps a tester to review the logic of the program under test.

- *strp* mutants:

  Distinguishing these mutants forces a tester to construct test cases which cover every executable statement in the program under test.

2.4 Uniqueness of experiments reported here

One major problem which occurred in the previous mutation studies was that the mutation tool **Mothra** only accepts Fortran programs. It was not unusual for experimenters to spend substantial effort and care in translating programs from a language such as C or Pascal to Fortran before any mutation experiments could be performed. Such translation might pose a danger of making some mutants unique to the Fortran 77 version, for they cannot be generated from the corresponding C or Pascal version of the same program. The reason for conducting such translation is that none of the current available data flow testing tools accepts Fortran programs. Although it has been shown that the effect of using different languages is negligible on the conclusions derived in the above studies [16], it is certainly desirable to avoid any language translation in the experiments. We consider our experiments reported here unique in that all constrained mutation was performed directly on C programs using **PROTEUM** [8], a C mutation tool implemented based on [2]. This has the advantage of eliminating any possible bias described above.

3 Comparison methodology

We compared the cost and effectiveness of adequate test sets with respect to various constrained mutation criteria. Effectiveness refers to the fault detection capability of a criterion. Cost refers to the *work* necessary to satisfy it. The cost of mutation testing can be measured in several ways. We selected two cost metrics. One is the number of test cases required to satisfy a criterion. As construction of each test case requires effort from a test case developer, this appears to be a reasonable cost metric. The second metric is the number of mutants to be examined. As mutants are to be executed on one or more test cases, a reduction in the number of mutants leads to a reduction in the time to execute them. It is also likely to result in a reduction in the time spent by a tester in examining mutants for possible equivalence.

Given a program P, a specification S, and a test set T adequate with respect to a criterion C, T is said to be able to expose faults in P if there exists a test case $t \in T$ which makes P behave differently from S. As long as T detects at least one fault in P, T is counted as a fault-revealing test set. Equation (1), defined below, is used to compute the percentage of C fault-revealing adequate test sets.

Table 1: Constrained mutations examined in our experiments

	Mutants examined
MUT-A	*olln, olng, orrn*
MUT-B	*olln, olng, orrn, ocng, orln, olrn, olan, oaln*
MUT-C	*vdtr, vtwd*
MUT-D	*strp*
MUT-E	*olln, olng, orrn, vdtr, vtwd*
MUT-F	*olln, olng, orrn, vdtr, vtwd, strp*

$$\frac{\text{number of } C \text{ adequate test sets which expose at least one fault}}{\text{number of } C \text{ adequate test sets generated}} * 100\% \quad (1)$$

4 Experimental methodology

We used a C mutation testing tool, **PROTEUM**, in our experiments. Given a program and a test set, **PROTEUM** generates a set of mutants, executes them against test cases in the test set, and computes the mutation score. A suite of five programs described below was selected. A source listing of each of these programs and their faults can be found in [20].

- **FIND**: This program takes two inputs, an array a and an index f, and permutes the elements of a so that elements to the right of position f are greater than or equal to $a[f]$ and elements to the left of position f are less than or equal to $a[f]$.
- **POSITION**: This program takes two inputs, an array a and a value *max*, and sums the elements of a until the sum meets or exceeds *max*. If such an element exists, its position is returned, otherwise a zero is returned.
- **SORT**: This program takes an array and sorts it in descending order.
- **STAT**: This program takes an array a and computes its sum, minimum, and maximum.
- **STRMATCH**: This program takes a text and a pattern of zero or more characters. If the pattern appears in the text then the position of the first occurrence of the pattern in the text is returned, otherwise a zero is returned.

Six different constrained mutations are examined based on the mutants discussed in Section 2.3. These mutations were labeled as indicated in Table 1 for reference. Three adequate test sets with respect to each constrained mutation criterion were randomly generated for each experiment listed in Table 2. Figure 1 shows the sequence of steps used in such generation. Multiple adequate test sets are necessary because for a given adequacy criterion there may exist an infinite number of test sets that satisfy it: selecting only one of these may possibly lead to

false conclusions. Such a possibility can be reduced by generating multiple test sets with respect to each criterion. All test cases were randomly selected, based on the uniform distribution unless otherwise specified[3], from the input domain described in Table 2. Such random generation is intended to eliminate any bias possibly introduced if human testers are used. This may happen, for example, when the testers are familiar with the programs used in the experiments and therefore generate test cases that favor one type of mutant over another. During the generation of various mutation adequate test sets, a test case was discarded if it could not distinguish at least one non-equivalent mutant. This requirement is intended to make a fair comparison among different criteria. In the absence of this requirement the fault detection effectiveness of a test set could be increased simply by including additional test cases. Note that our generation method only guarantees that each consequent test case is not redundant in terms of distinguishing mutants. It does not re-examine whether the previous test cases are still necessary. Once a test case is included in a test set, it will not be excluded due to the inclusion of any new test cases.

5 Experimental results and analysis

Table 3 lists the number of mutants examined in each constrained mutation. Tables 4 and 5 contain the average size of adequate test sets and the percentage of fault-revealing adequate test sets with respect to each constrained mutation.

Cost comparison

From our experimental data and the summary in Tables 3 and 4, we make the following observations:

- In terms of increasing order of the number of mutants to be examined the ranking is MUT-D, MUT-A, MUT-B, MUT-C, MUT-E, MUT-F.[4]
- For all ten experiments, criterion MUT-D requires the fewest number of test cases.
- In nine of ten experiments, criterion MUT-A requires the second fewest number of test cases.[5]
- Criteria MUT-C, MUT-E and MUT-F require about the same number of test cases which are, in general with some exceptions, more than criterion MUT-B.

Effectiveness comparison

From our experimental data and the summary in Table 5, we make the following observations:

[3] In EXPT-STRM1, STRM2, and STRM3, the probability of selecting a, b, and # is 5 : 5 : 1.

[4] The only exception occurs in program **SORT** which has MUT-A, MUT-D, MUT-B, MUT-C, MUT-E, MUT-F, in order.

[5] The only exception occurs in experiment **STRM3**.

Table 2: Experiment sets

Experiment	Program	Input domain†
FIND	FIND	• $-5 \leq$ array size ≤ 10 • $-10 \leq$ array element ≤ 100 • if array size > 0 then $1 \leq$ index $\leq$ array size else $-3 \leq$ index ≤ 2
POS1	POSITION	• array size=5 • $-5 \leq$ array element ≤ 10 • max $\in \{x \mid x = 5$ *or* $35 \leq x \leq 55\}$
POS2	POSITION	• array size=5 • array element $\in \{0, 1, 2, 3, 4, 5, 6, -5, 8\}$ • max $\in \{-5, 0, 1, 4, 8\}$
SORT1	SORT	• $-1 \leq$ array size ≤ 10 • $-10 \leq$ array element ≤ 100
SORT2	SORT	• $-1 \leq$ array size ≤ 3 • $-2 \leq$ array element ≤ 2
STAT1	STAT	• array size=5 • $-20 \leq$ array element ≤ 20
STAT2	STAT	• array size=5 • $-20 \leq$ array element ≤ 8
STRM1	STRMATCH	• $0 \leq$ text length ≤ 15 • $0 \leq$ pattern length ≤ 6 • *text element* $\in \{a, b, \#\}$ • *pattern element* $\in \{a, b, \#\}$
STRM2	STRMATCH	• $0 \leq$ text length ≤ 12 • $0 \leq$ pattern length ≤ 4 • *text element* $\in \{a, b, \#\}$ • *pattern element* $\in \{a, b, \#\}$
STRM3	STRMATCH	• $0 \leq$ text length ≤ 11 • $0 \leq$ pattern length ≤ 4 • *text element* $\in \{a, b, \#\}$ • *pattern element* $\in \{a, b, \#\}$

†All inputs are integers.

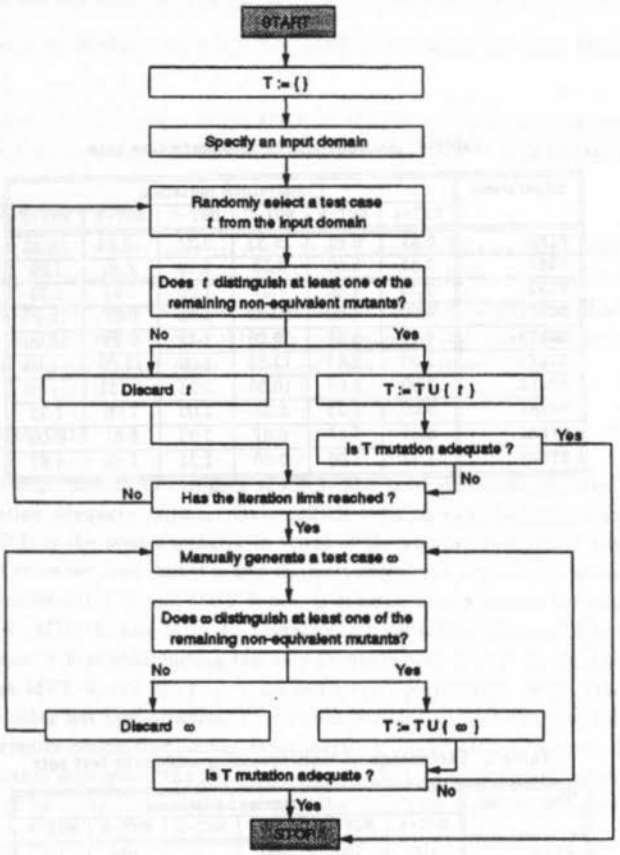

Figure 1: Procedure to generate various constrained mutation adequate test sets.

Table 3: Number of mutants examined

Program	Constrained mutation					
	MUT-A	MUT-B	MUT-C	MUT-D	MUT-E	MUT-F
FIND	39	80	185	33	224	257
POSITION	15	28	50	15	65	80
SORT	15	30	105	18	120	138
STAT	20	34	105	15	125	140
STRMATCH	24	56	80	16	104	120

Table 4: Average size of adequate test sets

Experiment	Constrained mutation					
	MUT-A	MUT-B	MUT-C	MUT-D	MUT-E	MUT-F
FIND	5.67	6.67	15.33	3.33	16.33	16.33
POS1	3.33	5.00	6.33	2.00	9.00	7.00
POS2	2.33	4.00	7.33	1.00	7.67	7.33
SORT1	4.00	4.00	10.33	1.33	9.67	10.00
SORT2	4.67	6.67	10.00	1.33	9.33	10.00
STAT1	2.00	2.67	12.33	1.00	13.00	11.00
STAT2	2.00	2.00	10.00	1.33	11.33	11.00
STRM1	5.00	5.33	5.33	2.00	7.00	7.33
STRM2	5.67	7.67	6.67	3.33	8.67	8.00
STRM3	7.33	7.00	6.00	2.33	7.00	6.67

Table 5: Percentage of fault-revealing adequate test sets

Experiment	Constrained mutation					
	MUT-A	MUT-B	MUT-C	MUT-D	MUT-E	MUT-F
FIND	100	100	100	0	100	100
POS1	100	100	100	33.33	100	100
POS2	100	66.67	66.67	66.67	66.67	100
SORT1	100	100	100	0	100	100
SORT2	100	33.33	100	0	100	100
STAT1	66.67	66.67	100	66.67	66.67	100
STAT2	66.67	66.67	66.67	0	100	66.67
STRM1	33.33	66.67	100	66.67	100	100
STRM2	0	0	100	0	100	100
STRM3	100	100	100	100	100	100

- In five of ten experiments, criterion MUT-D cannot detect any program fault.
- For experiment STRM2, criteria MUT-A, MUT-B and MUT-D cannot detect any program fault.
- In four of ten experiments, criteria MUT-A, MUT-B, MUT-C, MUT-E and MUT-F have the same fault detection effectiveness while criterion MUT-D is significantly less effective.

Effect of input domain

Multiple input domains were used to study their possible effect on fault detection effectiveness [19]. From our experimental data and the summary in Tables 2, 4 and 5, we find that to some extent varying the input domains has an impact on the relative order of adequate test set size and fault detection effectiveness of the six constrained mutation adequacy criteria discussed here. However, in general, such impact is insignificant in most situations.

6 Conclusions

We conducted experiments to compare the cost and fault detection effectiveness of six constrained mutation adequacy criteria. Data collected during experimentation have shown that criterion MUT-D is the least expensive in terms of the number of mutants examined and the number of test cases required, but it is also the least effective in exposing program faults. On the other hand, criteria MUT-A and MUT-B are, in general, less expensive but still as effective as criteria MUT-C, MUT-E, and MUT-F. Therefore, a good testing strategy is to adopt an incremental approach by first constructing test sets adequate with respect to the more cost-effective criteria such as MUT-A and MUT-B. Time and budget permitting, we may then improve test sets to satisfy other less cost-effective criteria such as MUT-C, MUT-E, and MUT-F.

We are currently conducting similar experiments on larger size C programs. Results of these studies will further strengthen the hypothesis that examining only a small carefully selected set of mutants may be a useful starting point for evaluating and constructing test sets.

References

[1] A. T. Acree, *"On mutation,"* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, August 1980.

[2] H. Agrawal, R. A. DeMillo, R. Hathaway, Wm. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. "Design of mutant operators for the C programming language." Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, W. Lafayette, IN, March 1989.

[3] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT – a formal system for testing and debugging programs by symbolic execution." *Sigplan Notices*, 10(6):234-245, June 1975.

[4] T. A. Budd. "*Mutation Analysis of Program Test Data.*" PhD thesis. Yale University. New Haven. CT. May 1980.

[5] T. A. Budd. "Mutation analysis: Ideas. examples. problems and prospect." in *Computer Program Testing.* B. Chandrasekaran and S. Radicchi. Eds. Amsterdam. North Holland. July 1981.

[6] B. J. Choi. R. A. DeMillo. E. W. Krauser. A. P. Mathur. R. J. Martin. A. J. Offutt. H. Pan. and E. H. Spafford. "The Mothra toolset." in *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, HI, January 1989.

[7] L. A. Clarke. "A system to generate test data and symbolically execute programs." *IEEE Trans. on Software Engineering*, SE-2(3):215-222. September 1976.

[8] M. E. Delamaro. J. C. Maldonado. M. Jino. and M. Chaim. "PROTEUM: A testing tool based on mutation analysis." in *Proceedings of the Seventh Brazilian Symposium on Software Engineering*, Rio de Janeiro. Brazil. October 1993.

[9] R. A. DeMillo. D. S. Guindi. K. N. King. W. M. McCracken. and A. J. Offutt. "An extended overview of the Mothra software testing environment." in *Proceedings of the Second Workshop on Software Testing. Verification and Analysis*, pp 142-151. Banff. Alberta. Canada. July 1988.

[10] R. A. DeMillo. R. J. Lipton. and F. G. Sayward. "Hints on test data selection: Help for the practicing programmer." *IEEE Computer*. 11(4):34-41. April 1978.

[11] W. E. Howden. "Reliability of the path analysis testing strategy." *IEEE Trans. on Software Engineering*. SE-2(3):208-214. September 1976.

[12] E. W. Krauser. A. P. Mathur. and V. J. Rego. "High performance software testing on SIMD machines." *IEEE Trans. on Software Engineering*, 17(5):403-423. May 1991.

[13] A. P. Mathur. "Performance. effectiveness. and reliability issues in software testing." in *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*, pp 604-605. Tokyo. Japan. September 1991.

[14] A. P. Mathur. "Mutation testing." *Encyclopedia of Software Engineering*, pp 707-713. 1994.

[15] A. P. Mathur and W. E. Wong. "Evaluation of the cost of alternate mutation testing strategies." in *Proceedings of the Seventh Brazilian Symposium on Software Engineering*, pp 320-335. Rio de Janeiro. Brazil. October 1993.

[16] A. P. Mathur and W. E. Wong. "An empirical comparison of data flow and mutation based test adequacy criteria." *The Journal of Software Testing. Verification. and Reliability*. 4(1):9-31. March 1994.

[17] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th International Conference on Software Engineering*, pp 100-107, Baltimore, MD, May 1993.

[18] S. N. Weiss and V. N. Fleyshgakker, "Improved serial algorithms for mutation analysis," in *Proceedings of International Symposium on Software Testing and Analysis*, pp 149-158, Cambridge, MA, June 1993.

[19] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. on Software Engineering*, 17(7):703-711, July 1991.

[20] W. E. Wong, "*On Mutation and Data Flow*," PhD thesis, Department of Computer Science, Purdue University, W. Lafayette, IN, December 1993.

[21] W. E. Wong and A. P. Mathur, "Fault detection effectiveness of mutation and data flow testing," *Software Quality Journal*, 1995. (To appear)

[22] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *The Journal of Systems and Software*, 1995. (To appear)

Appendix: Mutant operators used in the experiments

Operator	Meaning	Example†
OLLN	logical operator replacement	$(a > b) \,\|\|\, (c < d) \longrightarrow (a > b)\ \&\&\ (c < d)$
OLNG	logical negation	$(a > b) \,\|\|\, (c < d) \longrightarrow (a > b) \,\|\|\, !(c < d)$ $!(a > b) \,\|\|\, (c < d)$ $!((a > b) \,\|\|\, (c < d))$
ORRN	relational operator replacement	$a > b \longrightarrow a \geq b$ $a < b$ $a \leq b$ $a == b$ $a! = b$
OCNG	logical context negation	$a > b \longrightarrow !(a > b)$
ORLN	relational operator for logical operator replacement	$a > b \longrightarrow a\ \&\&\ b$ $a \,\|\|\, b$
OLRN	logical operator for relational operator replacement	$a > b \,\|\|\, c < d \longrightarrow a > b$ $a \geq b$ $a < b$ $a \leq b$ $a == b$ $a! = b$
OLAN	logical operator for arithmetic operator replacement	$a > b \,\|\|\, c < d \longrightarrow a + b$ $a - b$ $a * b$ $a \,/\, b$ $a \,\%\, b$
OALN	arithmetic operator for logical operator replacement	$a + b \longrightarrow a\ \&\&\ b$ $a \,\|\|\, b$
VDTR	domain trap	$s^{\ddagger} \longrightarrow TRAP\_ON\_POSITIVE(s)$ $TRAP\_ON\_NEGATIVE(s)$ $TRAP\_ON\_ZERO(s)$
VTWD	twiddling	$s^{\ddagger} \longrightarrow SUCC(s)$ $PRED(s)$
STRP	statement analysis	Each statement is replaced by $TRAP\_ON\_STMT()$

†$x \longrightarrow y$ means that string x in P is replaced by string y to obtain a mutant.

‡s represents a scalar reference.