Designing Deletion Mutation Operators

Marcio E. Delamaro*, Jeff Offutt†, Paul Ammann†
*Instituto de Ciências Matemáticas e de Computaçao, Universidade de São Paulo, São Carlos, SP, Brazil
†Software Engineering, George Mason University, Fairfax, VA, USA
Emails: delamaro@icmc.usp.br, {offutt,pammann}@gmu.edu

***Abstract*—As a test criterion, mutation analysis is known for yielding very effective tests. It is also known for creating many test requirements, each of which is represented by a "mutant" that must be "killed." In recent years, researchers have found that these test requirements have a lot of duplication, in that many test requirements yield the same tests. Put another way, hundreds of mutants can usually be killed by only a few dozen tests. If we could reduce this duplication without reducing mutation's effectiveness, mutation testing could become more cost-effective. One avenue of this research has been to use only one type of mutant, the statement deletion mutation operator. Researchers have found that statement deletion mutation has relatively few mutants, but yields tests that are almost as effective as using all mutants, with the significant benefit that fewer equivalent mutants are generated. This paper extends this idea by asking a simple question: if deleting statements is a cost-effective way to design tests, will deleting other program elements also be effective? This paper presents results from mutation operators that delete variables, operators, and constants, finding that indeed, this is an efficient and effective approach.**

I. Introduction

Mutation analysis [1] is a test criterion that is widely recognized for helping testers design very effective test sets. On the other hand, it is also considered costly, both in terms of computation and human resources. Mutation operators are used to generate alternative versions of the program under test and the tester designs tests that make these mutant programs behave differently from the original program. The number of mutants a test set "kills" is used as a measure of quality.

Mutation testing is effective because it directly relates to the ability of the test set to reveal faults, as represented by the mutants. Mutation adequate tests are likely to reveal other kind of faults, based on the coupling effect [2]. Mutation's effectiveness directly depends on the mutation operators used, as does its cost. It is expensive when the mutation operators create a large number of mutants (generally called test requirements [3]). Mutants must be run against the test set (a computation cost), and testers must analyze "live" mutants and design tests to kill them (a human cost). Additionally, some mutants cannot be killed, that is, they behave the same as the program under test for all tests; they are *equivalent* mutants. Identifying equivalent mutants is largely done by hand (a human cost). Not surprisingly, the human cost typically dominates.

Mutation operators play a pivotal role both the quality of the tests (effectiveness) and the cost of testing. Mutation operators are defined based on programming language characteristics and common mistakes programmers make. Mutation operators have been defined for many languages, including Fortran [4], C [5] and Java [6], [7]. Some mutation operators have been based on experience and intuition (Mothra operators [4]), whereas others have been based on fault models (muJava [7], and the Hazard of Operability Study (HAZOP) [8], [9]).

Mutation operators based on all characteristics of the language often generate mutants that are redundant in the sense that many mutants are killed by the same tests. Redundant mutants increases cost because there are more mutants, and some operators create more equivalent mutants than others.

Several studies were conducted to identify a smaller set of "essential" operators that would guarantee a good test set with a low cost [10]–[13]. A recent study [14] focused on a single operator with high effectiveness. The study found that the Statement Deletion (SDL) operator has characteristics that make it a likely choice for a single operator mutation criterion.

This paper uses that result, but goes in a new direction. Instead of empirically selecting mutation operators, we first analyze SDL characteristics and then develop new operators that should have the same qualities. This resulted in three new mutation operators that we applied to a suite of programs to assess effectiveness and cost. The results show that a combination of two operators is the most effective alternative, achieving more then 97% of mutation score gained when using the complete set of mutants. However, when cost is also taken in consideration, a more precise analyses is necessary. Thus, an analysis of cost is also presented.

This paper is organized as follows: The next section defines and discusses SDL-mutation, which uses only the SDL operator to generate mutants. Then we present our new operators. Section III presents an analysis of expected costs of all four operators. Section IV presents data from an experimental evaluation of the operators. Section V discusses similar work and Section VI gives final remarks and recommendations.

II. Deletion Mutation Operators

One-op mutation is the idea of using a single powerful mutation operator that leads to a highly effective test set with a low cost. SDL-mutation [15] is a one-op mutation version that uses Statement Deletion (SDL). Results showed that SDL-mutation yields huge savings in the cost of mutation testing without a significant loss of effectiveness [14].

SDL has several positive characteristics. First, its effectiveness is high. The usual way to assess an operator's effectiveness is to first design a test set that kills all of its mutants, and then find how many mutants that test set kills of the complete set of mutants. In previous experiments, SDL achieved 92% effectiveness for Java [14]. Second, the cost of using SDL is low. The total number of mutants is proportional to the number of statements in the program under test. In the worst case, if each mutant requires a unique test, the number of

tests is proportional to the number of statements. Third, the number of equivalent mutants is comparatively low. Statement deletion removes entire statements, so an SDL mutant is only equivalent if the statement is unnecessary. Previous work [14] identified a few, relatively rare, situations when SDL mutants could be equivalent (such as valid redundancy). However, the percentage of equivalent SDL mutants is less than that of the complete set of mutation operators.

Fourth, the SDL operator can be applied to every program, because every program must have at least one statement. This is not true for many other mutation operators. Fifth, SDL can be defined for any imperative programming language, making it broadly applicable. Sixth, the equivalent mutants created by SDL appear to be easier to identify than for other operators, manually or automatically. This is based on our observations, and the intuition that understanding the effects of removing a statement is easier than understanding the effects of other mutants. This observation has not been supported with data.

Our previous research demonstrated the value of SDL-mutation, making us wonder if other deletion operators might also be inexpensive and effective. Thus, this paper proposes and evaluates three additional deletion mutation operators. The idea is that removing structures from the program under test:

1) should generate mostly non-equivalent mutants, otherwise the structures could be eliminated from the original program
2) should generate comparatively few mutants, thus saving cost
3) should result in mutants for every unit under test
4) should be useful in most languages, by choosing structures to delete that are used in many languages
5) should generate comparatively few equivalent mutants that should be relatively easy to identify

To define new operators we considered the same major program structures modified by the original C mutant operators. They divide operators into four main categories, statement, operator, variable and constant. Statement deletion (SSDL) was already present, so we added operator deletion (OODL), variable deletion (VVDL), and constant deletion (CCDL).

They have been implemented in the C Proteum mutation system [16], using the same four-character naming convention. Proteum uses semantic analysis to avoid creating syntactically illegal (*stillborn*) mutants. Only about .1% mutants are stillborn, and they are filtered out by the compiler and thus are not used or counted. The following subsections define SSDL, OODL, VVDL, and CCDL. Although the discussion is in the context of C, the concept applies to other languages, including other paradigms that use similar structures. Sections III and IV complement this discussion with a theoretical analysis of cost and an experimental assessment of cost and effectiveness.

A. Statement Deletion

Statement deletion has been implemented in several mutation tools and languages. This study uses the C language tool **Pro**gram **Te**sting **U**sing **M**utants (Proteum) [16]. They follow, as closely as possible, the original C operators defined by Agrawal et al. [5]. Proteum's name for SDL is SSDL[1].

SSDL systematically removes each statement block, and each individual statement inside each statement block. SSDL systematically removes each statement as well as all inner statements. It does not change declarations, even when declarations include initialization assignments. Note that this is much more than statement coverage. Statement coverage only requires reachability [3], whereas SSDL requires an infection from the deleted statement and propagation to an output. The eleven mutants generated for the program in Figure 1(a) are described in Figure 1(b).

```
01 void test() {
02     int a, b, c, t, i;
03     if (a == 0) {
04         b = 3;
05     }
06     for (i = 0; i < 5; i++)
07         t = t + b + c;
08 }
09 void test_while() {
10     int a, b, c, t;
11     while (a < 5) {
12         t = t + b + c;
13         a++;
14     }
15 }
```

(a) Original program

	Lines removed
M1:	4
M2:	4, 5
M3:	3, 4, 5
M4:	7
M5:	6, 7
M6:	2, 3, 4, 5, 6, 7
M7:	12
M8:	13
M9:	12, 13, 14
M10:	11, 12, 13, 14
M11:	10, 11, 12, 13, 14

(b) SSDL mutants

Fig. 1. Examples of SSDL generated mutants.

An advantage of SSDL is the low number of mutants ($\mathcal{O}$(LOC)). Intuitively, one might expect no equivalent SSDL mutants unless the statement is unreachable or otherwise useless. That is, we might expect that all equivalent SSDL mutants represent something wrong with the program under test. Although this might often be true, some equivalent SSDL mutants might represent valid code. C compilers are permissive in many cases, allowing some unusual situations. Figure 2 shows two examples of equivalent SSDL mutants in C that do not represent problems with the program.

```
1 int foo (int j, int k) {
2     int i;
3     i = 0; // Equivalent SSDL mutant
4     // do something
5     i = k * j;
6     return i; // Equivalent SSDL mutant
7 }
```

Fig. 2. Equivalent SSDL mutants.

The first example is the mutant that deletes the initialization of variable i at line 3. This produces what we call a "quasi-equivalent" mutant. In C, the initial value of local variables that are not explicitly initialized is undefined: these variables are not set to a default value. The initial value of these

[1]For convenience, this paper uses "SDL" generically to mean an operator that deletes statements, and "SSDL" to refer to the C version of the operator. Thus, they are sometimes used interchangeably.

variables depends on the contents of the stack frame at function activation time. Therefore, the mutant in Figure 2 is not equivalent because it may be killed by chance, if a value different from zero is in the storage slot assigned to i. On the other hand, the tester is not able to provide input values that would kill this mutant. Mutants like this have to be considered equivalent, since the tester cannot design a test that is guaranteed to kill them.

The second example deletes the *return* statement, causing the function to return an unknown value. But if we analyze how the executable code is generated, the mutant is probably equivalent. For a particular compiler we may have the following sequence of instructions: (1) expression $k*j$ is computed in register R; (2) the value of R is stored in variable i; (3) value of i is moved to register R; (4) function foo returns its value on register R. Removing the *return* statement corresponds to not executing step (3) but the value of variable i is returned because it was already in R. So, with or without the *return* statement, the correct value is returned to the calling function through register R. Testers will not want to analyze at this level to mark mutants equivalent, so a reasonable approach is to treat mutants like this as equivalent.

In addition, other perfectly acceptable constructions may lead to equivalent SSDL mutants. For instance, an integer global variable is by default initialized with value zero. Nevertheless, it is a good practice to explicitly initialize the variable before using it. If the initial value is the same, then deleting such a statement may not change the behavior of the program (unless the function is called more than once and the variable changes values), leading to an equivalent mutant. In other words, redundancy is sometimes good engineering.

B. Operator Deletion

Part of the goal of this research is to identify other possible deletion mutation operators that can enhance SSDL. Our first attempt is a mutation operator that deletes C operators. OODL removes each arithmetic, relational, logical, bitwise and shift operator in expressions. It also removes them from assignment operators, replacing them by a plain assignment operator. When a binary operator is removed, an operand must also be removed so the expression remains well formed (compilable). Thus deleting a binary operator produces two mutants; one where the left operand is deleted, and another where the right operand is deleted. Figure 3 gives some examples of OODL mutants. The predicate in the example is shown fully parenthesized to emphasize the fact that, for example, when the '+' operator is removed, "`(2 * b)`" is the right operand (mutant M15).

As with SSDL, it is logical to expect that OODL would not create any equivalent mutants. If removing an operator (and part of the expression) never makes a difference, then the original expression must be incorrect. Still, some equivalent mutants are created that do not indicate an error, usually because of idiosyncrasies of C. Figure 4 shows some examples. In C, the expression `a != 0` is equivalent to `a`, so the first mutant does not change the program's behavior. Likewise, the expression `k` is equivalent to `k != 0`, which has the same effect in the loop test as `k > 0`. Both of these examples could be detected statically when the mutant is created, although our tool does not do this analysis.

Original statement	Mutant	
`if !((a + (2 * b)) > 0)`	M12	`(a + 2 * b > 0)`
	M13	`! (a + 2 > 0)`
	M14	`! (a + b > 0)`
	M15	`! (a > 0)`
	M16	`! (2 * b > 0)`
	M17	`! (a + 2 * b)`
	M18	`! ( 0 )`
x += 3 * y	M19	x += 3
	M20	x += y
	M21	x = 3 * y

Fig. 3. Examples of OODL generated mutants.

Original statement	Mutant
`if (a != 0)`	`if (a)`
`for (k=10; k>0; k--)`	`for (k=10; k; k--)`

Fig. 4. Equivalent OODL mutants.

Sometimes whether an OODL mutant is equivalent depends on dynamic aspects of the program. For example, deleting or changing a return statement whose value is never used is an equivalent mutant. Detecting these equivalent mutants requires much more analysis, either by a human or algorithm, and may be undecidable in some situations.

C. Variable Deletion and Constant Deletion

Variable (VVDL) and constant deletion (CCDL) operators are similar and discussed together. VVDL removes all occurrences of variable references from every expression, and CCDL removes all occurrences of constant references. They appear in expressions, thus the associated operator also must be removed.

Mutants M13, M16 and M19 in Figure 3 are VVDL mutants. Mutants M14, M17 and M20 are CCDL mutants. In fact, all VVDL and CCDL mutants are also OODL mutants, although OODL has additional mutants. For example, mutants M12, M15, and M18 are neither VVDL nor CCDL.

Defining mutants that are subsets of OODL allows us to measure the cost and effectiveness of mutation operators at a more detailed level. The VVDL and CCDL operators result in fewer mutants than OODL, thus are less expensive.

III. Cost Analysis

The cost of mutation testing is related to the mutation operators. Specifically, cost is influenced by the number of mutants created, the number of test cases required to kill the mutants, and the number of equivalent mutants. Cost can be assessed analytically or empirically.

An analytical approach can only address part of the cost. We can estimate the expected number of mutants generated by each operator, or a set of operators, but the actual number depends on the individual programs. In the worse case, the number of test cases required is the same as the number of mutants. This pessimistic analysis is far from actual, however, since each test case tends to kill many mutants. Equivalent mutants cannot be predicted as well, but can be estimated by empirically counting the number of equivalent mutants over a

collection of programs to arrive at an estimated percentage. This section presents an analysis of expected number of mutants for the proposed operators.

The number of mutants was initially estimated by Budd [17], who concluded that the number of mutants is $\mathcal{O}(Vars{*}Refs)$, that is, proportional to the number of variables in the program times the number of variable references. Thus, the number of mutants is dominated by the operators that replace variable references.

To the best of our knowledge, the complexity of the C operators has never been empirically measured. Delamaro et al. [18] evaluated C operators designed for integration errors, but did not include the traditional unit operators used in this paper. Because the same kind of replacement operators used in Mothra (Fortran 77) are also used in Proteum (C), it seems likely that the expression given by Budd is also valid for C.

The complexity of the SSDL operator is easy to compute. Each statement generates one mutant so the number of SSDL mutants is proportional to the number of lines of code, $\mathcal{O}(LOC)$. For the OODL operator, let's assume the maximum number of operators in a single expression is c. Then, in the worse case, the number of mutants is $2 * c * LOC$. If we assume that $c \ll LOC$ then we can treat c as a constant, and the complexity of OODL is also $\mathcal{O}(LOC)$. Since VVDL and CCDL are subsets of OODL, they are also bounded by the number of statements in the program.

IV. Experimental Analysis

This section presents and discusses the results of an experimental evaluation of the deletion operators. The goal is to completely evaluate the effectiveness and cost for each of the four operators, and then to evaluate possible combinations that could lead to a cost-effective set of mutants.

The following subsections describe the experimental setup, summarize the subject programs, present the results, discuss the data, and finally discuss threats to validity.

A. Experimental design

The goal of this study is to collect and analyze data about the effectiveness and cost of each deletion operator. First we collected this data with each operator in isolation. Then we investigated potential compositions of operators. Our experiment used:

- **Subjects:** 39 C programs from different sources and domains, and of different sizes. They are summarized in Subsection IV-B.
- **Independent variables:** The mutation operators used in the study: SSDL, OODL, VVDL, and CCDL, and combinations.
- **Dependent variables:** Mutation score, number of mutants generated, number of test cases required, and number of equivalent mutants.

The experiment was carried out in seven steps:

1) For each subject program, P_1, P_2, ..., P_{39}, all C mutation operators, new and old, were applied to generate the mutant set M_i.
2) For each M_i, a test set T_i was developed to kill all mutants, that is, $MS(T_i, M_i) = 1.0$. The tests were developed by hand by the first author. Equivalent mutants E_i were also hand identified.
3) For each deletion operator, we selected a subset of tests that killed all mutants of that type. That is, $MS(T_{i,j}, M_{i,j}) = 1.0$, where $T_{i,j} \in T_i$, $j \in \{SSDL, OODL, VVDL, CCDL\}$, and $M_{i,j}$ is the subset of mutants generated by operator j. So, $T_{i,j}$ is adequate to program P_i and operator j.
4) For each program P_i we measure the effectiveness of the deletion operator j by computing $MS(T_{i,j}, M_i)$, that is, the mutation score of the tests for operator j against the complete set of mutants.
5) For each program P_i we measure the cost of deletion operator j by computing the following:
   a) the percentage of all mutants generated by operator j: $\frac{|M_{i,j}|}{|M_i|} \times 100$
   b) the percentage of all tests needed to kill mutants by operator j: $\frac{|T_{i,j}|}{|T_i|} \times 100$
   c) the percentage of all equivalent mutants generated by operator j: $\frac{|E_{i,j}|}{|E_i|} \times 100$
6) For each deletion operator we measured the **weighted effectiveness**. This is computed by summing up the number of mutants killed by the test sets of operator j ($T_{i,j}$) and dividing this number by the total number of non equivalent mutants for all 39 programs.
7) We measured the **weighted costs** for each operator by summing up the costs of the operator for all 39 programs. Thus, for an operator j:
   a) weighted number of mutants: $\sum_{i=1}^{39} |M_{i,j}|$
   b) weighted number test cases: $\sum_{i=1}^{39} |T_{i,j}|$
   c) weighted number of equivalent mutants: $\sum_{i=1}^{39} |E_{i,j}|$

These measures allow us to compare the deletion operators. We also repeated the data collection with all combinations of the deletion operators.

In step 2, test cases were selected by hand to kill all non-equivalent mutants. Some programs already had tests, which were extended until they were mutation adequate. For the other programs, an ad-hoc strategy of analyzing mutants and designing tests to kill them was used.

In step 3, we selected tests from the complete adequate test set until all mutants from that mutation operator were killed. To avoid possible bias caused by the order of selection, we repeated this process 10 times, each time with a different random sequence of test cases. The mutation scores (number of mutants killed) and the number of test cases for each operator are averaged over these 10 different test sets.

B. Subjects

Thirty nine C programs of varying sizes and from different domains were used as experimental subjects. These programs were extracted from text books and the software testing literature. Program mutation is primarily used for unit testing, so we focused on program units (C functions) rather than large systems. The subject programs varied in size from one to 20

functions, and from seven to 394 lines of code, totaling 189 functions and 2853 lines of code.

Table I summarizes the subject programs. For each program, the table shows the number of functions, number of lines of code, the number of mutants generated by all operators, the total number of equivalent mutants, and the number of tests in the complete adequate test set.

TABLE I
EXPERIMENTAL RESULTS FOR SSDL MUTATION.

	Functions	LOC	Mutants	Equiv	Test cases
boundedQueue	6	49	1121	99	13
cal	1	18	891	71	8
Calculation	7	46	1118	107	13
checkIt	1	9	104	3	9
CheckPalindrome	1	10	166	20	8
countPositive	1	9	151	9	5
date-plus	3	132	2421	160	44
DigitReverser	1	17	496	43	5
findLast	1	10	198	17	6
findVal	1	7	190	18	7
Gaussian	6	23	1086	19	21
Heap	7	41	1079	98	8
InversePermutation	1	15	576	61	12
jday-jdate	2	49	2821	81	27
lastZero	1	9	173	9	5
LRS	5	51	1132	258	8
MergeSort	3	32	991	48	18
numZero	1	10	151	17	5
oddOrPos	1	9	361	71	7
pcal	8	204	6419	779	49
power	1	11	268	12	9
print_tokens	17	349	4322	542	34
print_tokens2	18	275	4734	664	27
printPrimes	2	35	715	64	7
Queue	6	64	469	25	12
quicksort	1	23	1026	82	13
RecursiveSort	1	17	555	45	8
replace	20	390	11,100	2062	142
schedule	18	213	2108	221	45
schedule2	16	195	2626	411	41
Stack	6	56	460	49	11
stats	1	19	884	101	7
sum	1	7	165	11	6
tcas	8	63	2384	428	62
testPad	1	24	629	57	14
totInfo	7	214	6693	678	49
trashAndTakeOut	2	19	599	26	12
twoPred	1	10	246	24	10
UnixCal	4	119	4852	339	27
Total	189	2853	66480	7829	814
Min	1	7	104	3	5
Max	20	390	11100	2062	142
Average	4.85	73.15	1704.62	200.74	20.87

C. Results

This section presents results, with discussion postponed to Section IV-D. Tables II through V show the statistics from applying each deletion operator to the subject programs for each dependent variable. These tables display unweighted results, that is, each of the 39 programs contributed to the results equally, regardless of whether it was a small program or a large program. Thus, the results in the tables are the average of the results for each individual program.

Table II shows the mutation scores of the test sets ($T_{i,j}$) when run against the complete set of mutants (M_i). That is, column SSDL shows results of the tests that kill all SSDL mutants when applied to all mutants; the mean mutation score across the 39 programs was .9232. We summarize the 39 programs by showing the lowest mutation score, the first quartile, the second quartile (mean and median), the third quartile, and the maximum mutation score. Table III shows the percentage of all mutants created by each operator. For example, averaged over the 39 programs, 3.608% of all mutants were SSDL. Table IV shows the percentage of all tests required to kill the mutants from each deletion operator. For example, on average 28.78% of all tests was needed to kill all SSDL mutants. Finally, Table V shows the percentage of all equivalent mutants that were created by each operator. So 2.021% of all equivalent mutants were of type SSDL. Figure 5 shows these data in boxplot graphs.

TABLE II
EFFECTIVENESS FOR EACH DELETION OPERATOR

Statistic	SSDL	OODL	VVDL	CCDL
Min.	0.6267	0.8531	0.7584	0.2712
1st Qu.	0.9072	0.9311	0.8855	0.7461
Median	0.9483	0.9598	0.9330	0.8737
Mean	**0.9232**	**0.9503**	**0.9172**	**0.8080**
3rd Qu.	0.9659	0.9770	0.9573	0.9322
Max.	0.9944	0.9912	0.9870	0.9861

TABLE III
MUTANTS COST FOR EACH DELETION OPERATOR

Statistic	SSDL	OODL	VVDL	CCDL
Min.	1.130	1.750	0.740	0.170
1st Qu.	2.235	2.415	1.140	0.420
Median	3.030	2.830	1.680	0.560
Mean	**3.608**	**2.947**	**1.618**	**0.658**
3rd Qu.	4.345	3.440	2.005	0.775
Max.	10.870	4.710	2.880	1.610

TABLE IV
TEST CASES COST FOR EACH DELETION OPERATOR

Statistic	SSDL	OODL	VVDL	CCDL
Min.	13.57	18.18	11.54	5.71
1st Qu.	21.63	24.61	19.86	10.00
Median	26.59	29.27	25.00	16.33
Mean	**28.78**	**33.43**	**26.86**	**17.21**
3rd Qu.	32.66	40.41	31.55	21.77
Max.	63.33	68.75	50.77	37.50

TABLE V
EQUIVALENT MUTANTS COST FOR EACH DELETION OPERATOR

Statistic	SSDL	OODL	VVDL	CCDL
Min.	0.000	0.00	0.00	0.00
1st Qu.	0.000	0.00	0.00	0.00
Median	1.230	1.980	0.00	0.470
Mean	**2.021**	**2.083**	**0.5651**	**1.045**
3rd Qu.	2.165	2.975	0.6150	1.770
Max.	18.370	12.240	5.000	4.650

We also collected data regarding possible combinations of these four operators. Not all 11 combinations were evaluated. VVDL and CCDL are subsets of OODL, so combining either with OODL will by definition not increase effectiveness. Thus, instead of six two-way combinations, four three-way combinations and one four-way combination, we only evaluated four two-way combinations, plus the three-way combination of SSDL, VVDL, and CCDL. We designate the combinations by combining the first letters of the operators' names:

- SODL = SSDL ∪ OODL
- SVDL = SSDL ∪ VVDL
- SCDL = SSDL ∪ CCDL

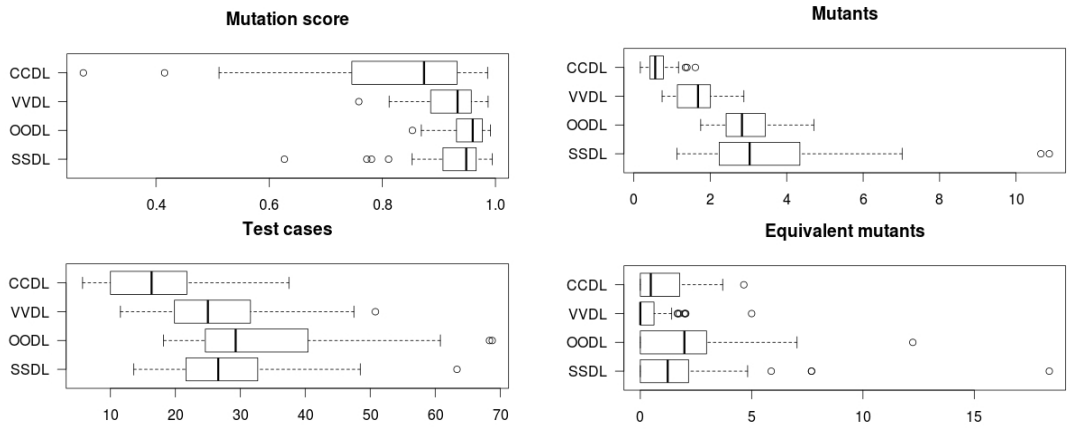

Fig. 5. Boxplot of effectiveness and cost for individual operators

- VCDL = VVDL ∪ CCDL
- SVCDL = SSDL ∪ VVDL ∪ CCDL

The data from these combinations are shown in Tables VI through IX, and their boxplot graphs are in Figure 6.

TABLE VI
EFFECTIVENESS FOR COMBINATIONS OF OPERATORS

Statistic	SODL	SVDL	SCDL	VCDL	SVCDL
Min.	0.8689	0.8287	0.6267	0.7584	0.8287
1st Qu.	0.9558	0.9372	0.9351	0.9118	0.9506
Median	0.9751	0.9621	0.9593	0.9450	0.9696
Mean	**0.9666**	**0.9525**	**0.9391**	**0.9339**	**0.9607**
3rd Qu.	0.9837	0.9790	0.9748	0.9706	0.9797
Max.	0.9948	0.9944	0.9944	0.9870	0.9944

TABLE VII
MUTANTS COST FOR COMBINATIONS OF OPERATORS

Statistic	SODL	SVDL	SCDL	VCDL	SVCDL
Min.	3.900	1.990	1.800	1.230	3.230
1st Qu.	4.965	3.790	2.835	1.865	4.245
Median	6.100	4.740	3.730	2.210	5.310
Mean	**6.555**	**5.226**	**4.200**	**2.209**	**5.817**
3rd Qu.	7.280	6.000	4.865	2.590	6.775
Max.	15.140	13.220	11.09	3.240	13.430

TABLE VIII
TEST CASES COST FOR COMBINATIONS OF OPERATORS

Statistic	SODL	SVDL	SCDL	VCDL	SVCDL
Min.	20.00	16.57	15.52	13.64	17.41
1st Qu.	29.00	26.20	24.18	21.62	28.45
Median	36.67	32.00	28.52	27.14	34.29
Mean	**38.63**	**34.71**	**31.91**	**30.01**	**36.89**
3rd Qu.	45.40	38.01	39.38	34.50	43.47
Max.	68.75	65.00	65.83	65.00	68.33

TABLE IX
EQUIVALENT MUTANTS COST FOR COMBINATIONS OF OPERATORS

Statistic	SODL	SVDL	SCDL	VCDL	SVCDL
Min.	0.00	0.00	0.00	0.00	0.00
1st Qu.	0.00	0.00	0.00	0.00	0.00
Median	3.750	1.560	2.060	1.560	3.120
Mean	**4.105**	**2.587**	**2.959**	**1.503**	**3.525**
3rd Qu.	5.130	3.735	4.545	2.385	4.970
Max.	30.610	20.410	18.370	5.000	20.410

To compute the cost of each operator, we use the percentage of test cases it requires and the percentage of equivalent mutants. The number of tests and equivalent mutants affect human cost. The number of mutants is not considered in the cost because it only affects computation.

Table X shows a summary of effectiveness and cost for each deletion operator both by itself and in combination with other deletion operators. To enable a comparison with traditional mutation, we also included the same results for random subsets of mutants. These subsets varied from 1% to 5% of the complete set of mutants for each program. We call these "*% random selective mutation*." The values were chosen so costs, in terms of mutation score and number of test cases, and the benefit, in terms of mutation score, were in the same range as the results for the delete mutation operators. This allows us to compare the approaches using the graphs in Figures 7 and 8. The CCDL operator, which scored poorly overall relative to the other operators, is not shown in the plots in Figures 7 and 8, thus making the remaining results easier to see.

To reduce variation in the results obtained for % random selective mutation, the data for these criteria were collected five times, with different sampling of mutants. The results presented average over those five samples.

To compare two criteria using Figure 7 or 8, note their relative positions on the plot. The vertical axis measures the mutation score, so it is better to be higher on the plot. The horizontal axis measures the cost of either more tests or more equivalent mutants to examine, so being farther left on a plot is also desirable. Hence, if one criterion plots both above and to the left of a second, the first criterion outperforms the second. If the two criteria have some other relation on the graph, then definitive conclusions cannot be drawn.

So far, we have presented unweighted averages over the 39 programs. This means that small programs of 9 or 10 LOC have the same influence on the average as large programs with two or three hundred LOC. This has the advantage of straightforward and unbiased but means the results can be unduly influenced by small outliers.

The results from individual programs can also be combined by weighting the totals based on the program sizes. The simplest way is to add the number of mutants, tests, mutants killed, and equivalent mutants, then compute mutation score and cost based on these sums. This "average of sums" reduces

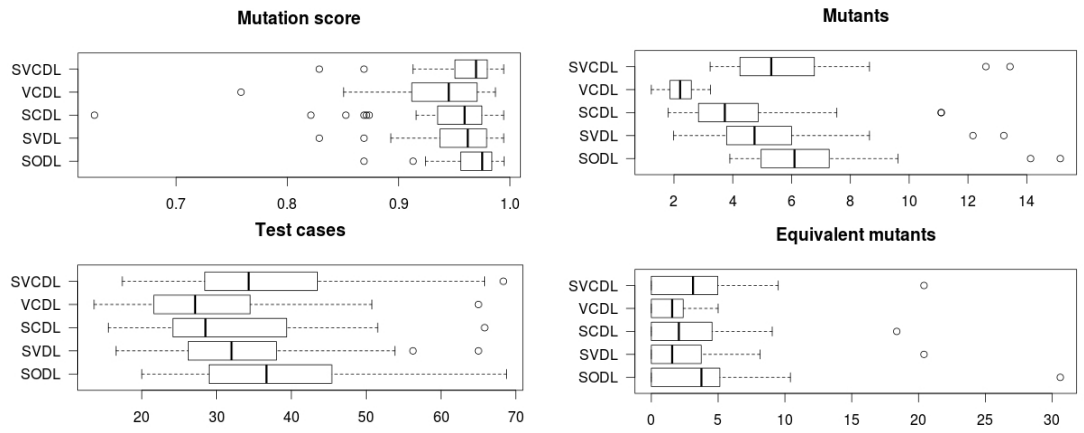

Fig. 6. Boxplot of effectiveness and cost for combinations of operators

TABLE X
EFFECTIVENESS AND COST RESULTS.

Operator	MS	%T. C.	%Equiv
CCDL	0.8080	17.21	1.04
OODL	0.9503	33.43	2.08
SSDL	0.9232	28.78	2.02
VVDL	0.9172	26.86	0.57
SCDL	0.9391	31.91	2.96
VCDL	0.9339	30.01	1.5
SODL	0.9666	38.63	4.11
SVDL	0.9525	34.71	2.59
SVCDL	0.9607	36.89	3.52
Random 1%	0.8618	22.91	1.20
Random 2%	0.9250	30.45	2.45
Random 3%	0.9419	34.24	3.24
Random 4%	0.9539	38.72	4.37
Random 5%	0.9611	41.66	5.23

the impact of small programs, and is shown in Table XI, with graphs in Figure 8.

TABLE XI
WEIGHTED EFFECTIVENESS AND COST RESULTS.

Operator	MS	# T. C.	# Equiv
CCDL	0.8664	105.8	68
OODL	0.9661	228.7	186
SSDL	0.9552	202.6	175
VVDL	0.9376	169.5	54
SCDL	0.9666	227.5	243
VCDL	0.9471	189.9	122
SODL	0.9806	278.2	361
SVDL	0.9718	243.8	229
SVCDL	0.9754	257.3	297
Random 1%	0.9194	153.62	75.20
Random 2%	0.9517	205.80	154.00
Random 3%	0.9660	237,70	238.00
Random 4%	0.9739	272.48	304.80
Random 5%	0,9791	294.38	387.40

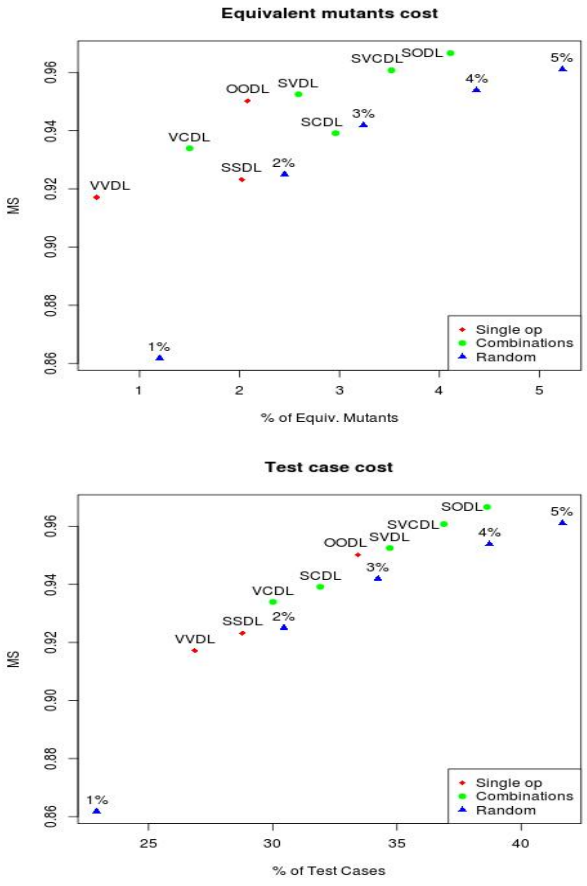

Fig. 7. Unweighted cost and effectiveness graphs.

To complete the analysis, the values of the mutation scores obtained by each criterion were compared in a Wilcoxon paired test to check the following hypotheses:

- Null hypothesis, H_0: there is no significant difference between criteria A and B in the values of the MS over the 39 programs.
- Alternative hypothesis, H_1: there is a significant difference between criteria A and B in the values of the MS over the 39 programs.

Table XII shows one line for each criterion A and one column for criterion B. The shaded cells highlight the cases in which the Deletion Operators have a higher mutation score than the random criterion. The pairs that did not have significant improvement at the 95% confidence level are not shown. For instance, the first line shows that SSDL has a p-value of $2e-07$ when compared with the Random 1% criterion, thus we reject H_0 in favor of H_1 and SSDL has a better score. There is no significant difference between SSDL and Random 2%, and Random 3%, 4% and 5% have better

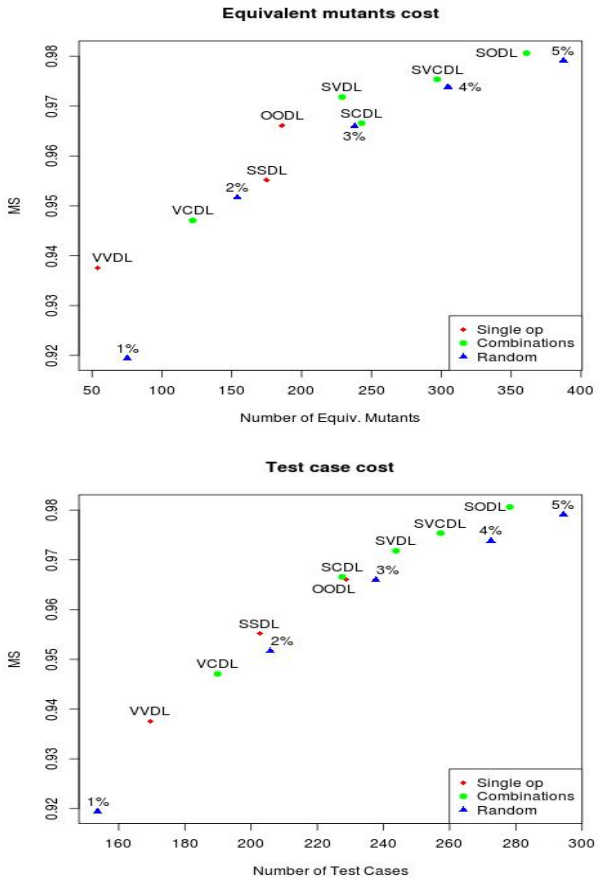

Fig. 8. Weighted cost and effectiveness graphs.

TABLE XII
P-VALUES FOR THE WILCOXON TEST BETWEEN MS CRITERIA.

	1%	2%	3%	4%	5%
SSDL	2e-07	–	0.0030	4e-06	5e-08
OODL	1e-10	0.0002	–	–	0.0013
VVDL	4e-05	–	7e-06	5e-08	3e-12
CCDL	–	7e-05	5e-07	3e-09	1e-10
SODL	3e-11	1e-06	1e-06	0.0120	–
SVDL	6e-11	8-06	–	–	0.0028
SCDL	1e-09	0.0246	–	0.0012	7e-06
VCDL	2e-08	–	0.0222	0.0002	2e-06
SVCDL	3e-11	1e-06	8e-05	–	–

mutation scores than SSDL.

D. Discussion

If we only use one or two mutation operators, it is important that they apply to all programs. This is trivially true for SSDL (every program has at least one statement) and is usually true for the other operators in this paper since they mutate basic language features. Nevertheless, four of the 39 programs produced no CCDL mutants. These four program, *checkIt*, *checkPalindrome*, *findVal* and *sum*, are very small (7 to 10 LOC) and simple. *checkPalindrome* has one constant, but it is used in a variable declaration and initialization, which Proteum does not mutate.

1) Effectiveness: SSDL has been found to be particularly effective [14], [15]. In this experiment it averages a 0.9232 mutation score, with a median of 0.9483. It does not perform well for the smallest programs, having mutation score under 0.8 for three. For these programs, SSDL creates only a few mutants (six or seven) that are killed by only two or three tests. They have very few mutants (from 104 to 246), so a small difference in the number of killed mutants makes a big difference in the mutation scores. If those outliers were eliminated, SSDL's average mutation score would be 0.9396.

VVDL also exhibits the same effect, with one program with mutation score of under 0.8. If that program was not used, VVDL's mean mutation score would be 0.9213.

OODL has the highest mean mutation score (.9503) and median (.9598), as well as the smallest variation. Its minimum score of 0.8531 could be considered an outlier. CCDL has very low mutation scores (0.8080 average), and so should not be considered as an effective single operator choice.

Combining operators yields higher mutation scores, with the combination of SSDL and OODL (SODL) as the highest. Its average mutation score is 0.9666, with a median of 0.9751, has small variation, and has a minimum score of 0.8689. So if we consider only effectiveness, SODL seems the best choice.

Other combinations also look strong. The combination of SSDL and VVDL (SVDL) have a mean mutation score of 0.9525, and a median of 0.9621. The mutation score is below 0.8 for one program, and under 0.9 for three others. Adding the CCDL operator (SVCDL) raises these numbers so they are comparable with SODL.

Combining SSDL with CCDL (SCDL) or VVDL with CCDL (CVDL) yields high scores, although not quite as high: 0.9391 and 0.9339 average with medians of 0.96 ad 0.97. SCDL also has a very low minimum of 0.6267, which accounts for its relative high variation. This program is an outlier because CCDL does not generate any mutants and SSDL produces poor results. Without this outlier the minimum for SCDL would be 0.8112.

2) Cost: The other three dependent variables in the experiment relate to the cost of mutation testing. Perhaps the most encouraging results are with equivalent mutants. Only 0.57% of the equivalent mutants are of type VVDL on average, and the program with the most has only 5%. In fact, more than 50% of the programs had zero equivalent VVDL mutants, that is, the median is 0.0%. CCDL has similar numbers, and not surprisingly, VCDL is a promising combination.

The percentages of mutants generated are directly related to the structure affected by the operators. One unexpected result is that OODL generates, on average, fewer mutants than SSDL. Since each statement may have more than one operator and each binary operator produces two mutants, we expected OODL to have more, but in our subjects, OODL created 2.95% of the mutants and SSDL 3.61%.

VVDL created fewer mutants than SSDL and OODL, and CCDL even fewer. The number of mutants for the combinations is the sum of the individual operators, with a linear ordering of: VCDL $<$ SCDL $<$ SVDL $<$ SVCDL $<$ SODL. Only SSDL showed a large variability in the percentage of mutants generated, and generated over 10% of the mutants for two (very small) subjects.

The percentage of test cases needed to kill the mutants of each operator showed surprising variation. CCDL required

the fewest tests on average, but had the most variation (from 5.71% to 37.50%). The combination of SSDL and OODL (SODL) was the most expensive, averaging 38.63% with a maximum of 68.75%. The ordering among the operators is similar to that with mutants, except that SSDL needed fewer tests than OODL.

3) Summary: Based on these data, the best operator (or combination) depends on whether cost or effectiveness is more important.

If effectiveness is key, then SODL is the best choice. If the number of mutants (cost) is the most important, then VVDL is best. Quite often however, testers want a solution that would balance cost and effectiveness. Tables X and XI summarize the information about cost and effectiveness and give some comparison data, using 1% to 5% random criteria. Figures 7 and 8 allows a visual analysis of such data.

We can see that when comparing operators with similar cost, the effectiveness will usually favor the deletion operators over the random criteria, especially when considering the cost of identifying equivalent mutants. Using the unweighted data, OODL's equivalent mutant cost is less than Random 2% but it has a higher mutation score; even higher than (or comparable with) Random 3%'s. SVDL has almost the same cost as Random 2% and a higher effectiveness. In fact, SVDL's mutation score is higher than (or comparable with) Random 3%'s and similar to Random 4% but with a lower percentage of equivalent mutants. The same analysis can be done pairing SODL and SVCDL with Random 4% and Random 5%. One remarkable case is VVDL, which has a cost of less than 0.6% of equivalent mutants but still an effectiveness above 91%.

Using the weighted equivalent mutant cost, the conclusions are also similar. OODL is stronger than Random 3%, with a lower cost. SVDL is comparable with Random 4% but significantly lower cost. SVCDL is comparable with Random 4% and SODL is better than Random 5%.

The same analysis holds for the test case cost, but some of the differences are smaller. As a visual aid, looking at graphs we can draw an imaginary crescent curve passing through the random criteria points. Assuming this curve represents the behavior of all random criteria, it is noticeable that all deletion operators are above the curve, indicating that they provide better effectiveness and/or lower cost.

E. Threats to validity

This experiment used a relatively large number of C programs, collected from diverse sources and with diverse features. As with any experimental study using programs, however, we cannot be sure whether they are representative, which limits the results' generalizability.

Another common problem in this kind of experimental work is the need to obtain an adequate test set for the programs. This can be done automatically, for instance, generating a large set of test cases and assuming that all the remaining alive mutants are equivalent. While this approach can save time, it would introduce imprecision. Thus, in this experiment we manually analyzed each mutant and either created a test to kill it or identified it as equivalent. This is tedious and time consuming, but can reduce error. This also made building more than one test set per program prohibitively expensive.

The potential error associated with selecting test sets adequate to the operators was minimized by selecting, for each program, 10 different sets from the original complete test set, for each target operator.

V. Related Work

Several papers have provided details about defining mutation operators. Most widely known are mutation operators defined for Fortran [4], [19] (implemented in Mothra), C [5] (in Proteum), and Java [7] (in muJava).

This paper was inspired by previous work that attempted to use a single operator to reduce the cost of mutation testing, in particular, the Statement Deletion operator. Untch carried out an experiment across four sufficient sets of mutation operators and the single statement deletion operator [15]. He used regression analysis to show that SDL generates the fewest mutants, and was best at predicting the overall mutation score of the test sets. Similar results were achieved by Deng et al. [14], who showed that SDL can dramatically reduce the cost of the tests while still being effective.

Another example of mutation operators designed for specific types of errors are the Interface Mutation (IM) operators [20]. Those operators reveal integration faults by mimicking faults related to errors that propagate through a function call, its parameters, and return value.

Kaminski et al. took a more theoretical approach to reduce redundancy among mutation operators at time of creation. They proved that the relational operator replacement operator, which normally produces seven mutants for each operator, only needs three mutants per operator. Tests that kill those three are guaranteed to kill the other four. Just, Kapfhammer, and Scheiggert [21] obtained similar results for the conditional operator replacement mutation operator.

VI. Conclusions

This paper presents new results to reduce the cost of mutation testing. The statement deletion mutation operator (SSDL) deletes entire statements from programs, thus requiring the tester to design tests that demonstrate the usefulness of each statement. This paper defines new innovative mutation operators that delete parts of statements, and presents results from an empirical evaluation of the new operators. Deletion mutation operators allow testers to achieve most of the benefits of traditional mutation testing at a fraction of the cost.

The operators were designed to: (1) require test sets that are highly effective at revealing faults; (2) generate relatively few mutants; (3) generate relatively few equivalent mutants; (4) apply to all programs; (5) be applicable to diverse programming languages; (6) have equivalent mutants that are easy to identify. The statement deletion operator satisfies these goals, and we found that the new deletion operators do as well.

This paper defines three operators. Operator deletion (OODL) removes arithmetic and relational operators, including each of the left and right operands of binary operators (two mutants). Variable deletion (VVDL) removes variable

references, including operators when needed. Constant deletion (CCDL) removes constant references, including operators when needed.

All operators create $\mathcal{O}(LOC)$ mutants, that is, the number of mutants is linear in the number of statements. This compares very favorably with $\mathcal{O}(Vars * Refs)$ for the traditional set of mutation operators. And unlike other mutation operators, they will generate mutants for every program (at least, every program that contains statements, variables, and operators). They also can be applied to any programming language that includes statements, variables, and operators.

Our experimental results showed that the OODL and VVDL mutation operators perform remarkably well. Tests that kill all OODL and VVDL mutants kill a very high percentage of all mutants, indicating the test sets are almost as effective. We cannot quantify how much testing strength we lose by achieving 97% mutation score instead of 100%, but this is significantly more testing than is normally achieved in practice. CCDL was less effective. We also combined the new operators (and SSDL), finding that the combination of SSL and OODL is very effective as well as very inexpensive.

Very importantly, we found that the deletion operators needed fewer tests than the traditional set of operators, and generated a fraction of the number of equivalent mutants. Both of these are significant factors in the cost of mutation testing. These results were obtained with the C programming language (using Proteum), and many equivalent mutants were only equivalent because of unusual characteristics of the language. If this study was redone in Java, we expect even fewer equivalent mutants.

We did not empirically evaluate whether it is easier to detect equivalent deletion mutants, so have no quantitative data. However, we hand-identified hundreds of equivalent mutants in this study, and observed that most equivalent deletion mutants were very easy to confirm, whereas many equivalent non-deletion mutants required very detailed and time-consuming analysis. Intuitively, this is because the deletion mutants are fairly simple, and their affects on program behavior are usually clear and straightforward.

These results will benefit other areas of testing research like automatic test data generation [22], [23]. The authors are currently working the generation of test data based on mutation. Narrowing the focus to only a few, relatively simple mutation operators greatly simplifies the problem.

ACKNOWLEDGMENTS

Prof. Marcio Delamaro's research is supported by FAPESP (Fundação de Amparo a Pesquisa do Estado de São Paulo, Brazil), process number 2012/16950-5 and CNPq process number 401745/2013-9.

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[2] J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering Methodology*, vol. 1, no. 1, pp. 3–18, January 1992.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2008, iSBN 0-52188-038-1.

[4] K. N. King and J. Offutt, "A Fortran language system for mutation-based software testing," *Software-Practice and Experience*, vol. 21, no. 7, pp. 685–718, July 1991.

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

[6] S. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented strategies with the mutation method," in *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October 2000, pp. 4–100, Wiley's Software Testing, Verification, and Reliability, December 2001.

[7] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava : An automated class mutation system," *Software Testing, Verification, and Reliability, Wiley*, vol. 15, no. 2, pp. 97–133, June 2005.

[8] S. Kim, J. A. Clark, and J. A. McDermid, "The rigorous generation of Java mutation operators using HAZOP," University of York, Tech. Rep., 2000.

[9] M. E. Delamaro, M. Pezze, V. Auri M. R, and J. C. Maldonado, "Mutant operators for testing concurrent java programs," in *XV Brazilian Symposium on Software Engineering*. Brazilian Computer Society, 2001, pp. 272–285.

[10] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Constrained mutation in C programs," in *Proceedings of the 8th Brazilian Symposium on Software Engineering*, Curitiba, Brazil, October 1994, pp. 439–452.

[11] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software, Elsevier*, vol. 31, no. 3, pp. 185–196, December 1995.

[12] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An experimental determination of sufficient mutation operators," *ACM Transactions on Software Engineering Methodology*, vol. 5, no. 2, pp. 99–118, April 1996.

[13] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Software Testing, Verification, and Reliability, Wiley*, vol. 11, pp. 113–136, 2001.

[14] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, Luxembourg, March 2013.

[15] R. Untch, "On reduced neighborhood mutation analysis using a single mutagenic operator," in *ACM Southeast Regional Conference*, Clemson SC, March 2009, pp. 19–21.

[16] M. E. Delamaro and J. C. Maldonado, "Proteum-A tool for the assessment of test adequacy for C programs," in *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, New Brunswick, NJ, July 1996, pp. 79–95.

[17] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, New Haven CT, 1980.

[18] M. E. Delamaro, "Interface mutation: an interprocedural criterion for integration testing," Ph.D. dissertation, IFSC – Universidade de São Paulo, São Carlos, SP – Brazil, 1997, in Portuguese. [Online]. Available: http://www.teses.usp.br/teses/disponiveis/76/76132/tde-26112008-130813/

[19] R. A. DeMillo and J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[20] M. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, March 2001.

[21] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *Eighth Workshop on Mutation Analysis (IEEE Mutation 2012)*, Montreal, Canada, April 2012.

[22] J. Offutt, "An integrated automatic test data generation system," *Journal of Systems Integration*, vol. 1, no. 3, pp. 391–409, November 1991.

[23] J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction approach to test data generation," *Software-Practice and Experience*, vol. 29, no. 2, pp. 167–193, January 1999.