Experimental evaluation of SDL and One-Op mutation for C

2014 IEEE International Conference on Software Testing, Verification, and Validation

Experimental Evaluation of SDL and One-Op Mutation for C

Marcio E. Delamaro,* Lin Deng,† Vinicius H. S. Durelli,* Nan Li,† and Jeff Offutt†
*Computer Systems Department
Universidade de São Paulo, São Carlos, SP, Brazil
Emails: {delamaro,durelli}@icmc.usp.br
†Software Engineering
George Mason University, Fairfax, VA, USA
Emails: {ldeng2,nli1,offutt}@gmu.edu

***Abstract*—Mutation analysis modifies a program by applying syntactic rules, called mutation operators, systematically to create many versions of the program (mutants) that differ in small ways. Testers then design tests to cause the mutants to behave differently from the original program. Mutation testing is widely considered to result in very effective tests, however, it is also quite costly. Cost comes from the many mutants that are created, the number of tests that are needed to kill the mutants, and the difficulty of deciding whether mutants behave equivalently to the original program. One-op mutation theorizes that cost can be reduced by using a single, very powerful, mutation operator that leads to tests that are almost as effective as if all operators are used. Previous research proposed the statement deletion operator (SDL) and found promising results. This paper investigates the use of SDL-mutation in a new context, the language C, and poses additional empirical questions, including whether other operators can be used. We carried out a controlled experiment in which cost and effectiveness of each individual C mutation operator were collected for 39 different subject programs. Experimental data are used to define a cost-effectiveness metric to choose the best single operator for one-op mutation.**

***Index Terms*—Software testing; Mutation testing; Mutation operators; SDL-mutation; One-op mutation**

I. Introduction

Program mutation analysis or mutation testing [4, 5] creates modified versions of programs, called *mutants*. Mutants contain small syntactic changes that mimic possible programmer mistakes or that encourage specific high quality tests. Testers then design inputs that *kill* the mutants by causing them to have different behaviors from the original program. Mutants that always have the same behavior as the original program and are called *equivalent* mutants. The syntactic changes that generate mutants are defined by *mutation operators*. Thus, mutation operators play a pivotal role in helping testers design high quality tests. Well-designed mutation operators can lead to effective testing whereas poorly designed operators can result in ineffective tests. A *mutation score* is a ratio of killed mutants over all non-equivalent mutants. Mutation scores often are used to evaluate the effectiveness of test sets, but this paper uses the same tests with different mutation operators to evaluate the effectiveness of the operators.

Mutation operators have been defined for several programming languages, including Fortran 77 [4, 14], C [3], and Java [13, 17]. The mutation operators that modify individual statements (*statement-level operators*) have remained relatively stable since the Mothra project [14], with the major change being from the *selective* operator study [26]. That study found that using five mutation operators in Mothra produced tests that kill most other mutants. Most mutation systems since have used operators based on the selective set.

Mutation analysis helps testers design very good tests, but it creates many test requirements (that is, mutants) as compared to other test criteria. Several papers have reported on the number of mutants.

Budd [2] found that the number of mutants is roughly proportional to the product of the number of variable references times the number of data objects: Offutt et al. [26] used a statistical regression analysis of actual programs to show that the number of lines did not contribute to the number of mutants, and confirmed Budd's equation. That paper also introduced *selective mutation*, defining five Mothra operators for which the number of mutants is proportional to the number of variable references: $O(Refs)$.

Although selective mutation generates far more test requirements than the edge-pair, all-uses, and prime path criteria, Li et al. [16] found that it needs fewer tests. An important implication of this finding is that many mutants are redundant, leading us to postulate that mutation testing can be effective with fewer mutants.

The hypothesis in this study is tests that kill all mutants of a single mutation operator can be effective at killing "almost" all the other mutants, at a much lower cost. We refer to this approach as *one-op mutation*.

Deng et al. [7] found that the SDL (statement deletion) operator works well. SDL generates relatively few mutants and leads to tests that are highly effective at killing other mutants. This paper extends that work by first reproducing the experiment for the C language, and then extending it by considering all other C mutation operators as possible choices. Based on these experimental results, we propose a metric to choose between mutation operators. Such a metric takes effectiveness and cost into consideration to select the best single operator to use.

The remainder of this paper is organized as follows. Sec-

978-0-7695-5185-2/14 $31.00 © 2014 IEEE
DOI 10.1109/ICST.2014.33

tion II presents previous work that attempts to reduce the cost of mutation and in particular SDL-mutation. Section III discusses details of the SDL operator for the C language. Section IV presents the experimental setup that evaluates the characteristics of the SDL operator and then compares it with the other operators for C. Section V discusses the results and threats to validity, and Section VI presents final remarks and recommendations.

II. Background and Related Work

The statement deletion operator was included in the first mutation tools, including Mothra [4]. It has also been designed for other languages, including C [3] and Java [7].

Offutt and Untch [24] categorized approaches to reduce the computational cost of mutation into three general strategies: *do-fewer*, *do-smarter*, and *do-faster*. Do-fewer approaches try to reduce the number of mutants that need to be run without losing much effectiveness. Do-smarter approaches distribute the computational expense over several computers or over multiple executions by storing run-time information between runs. Do-faster approaches speed up the generation and execution of mutants.

Mutant sampling is a do-fewer approach. Wong [30] suggested a straightforward technique: randomly select a subset of all mutants according to a uniform distribution. He found that the resulting tests were significantly weaker when the sampling rate was low enough to yield significant savings.

Selective mutation [31, 32] uses the most critical mutation operators. The idea was investigated by Offutt et al. [26], who found that Mothra's mutation operators could be reduced from 22 to five operators and still achieve almost the same test strength as non-selective mutation.

Namin et al. [20] expanded selective mutation by adapting statistical techniques to support the identification of selective sets. They started with an initial set of 108 C mutation operators, and used their procedure to reduce to 28 operators, a reduction of 92% in the number of mutants. Rather than only looking at how well the selective set performs, their approach also looks at the entire range of possible scores.

Kaminski et al. [12] investigated a do-fewer approach. The *relational operator replacement* mutation operator (ROR) creates seven mutants per relational operator. Kaminski et al. showed that only three of these seven mutants are needed: tests that kill these three mutants are guaranteed to kill the remaining four. Just et al. had similar results with the *conditional operator replacement* (COR) mutation operator [11].

Most do-smarter approaches have used advanced computer architectures to distribute the computational expense of running mutants among several processors. Each mutant execution is independent, so this problem is well-suited to parallelism. Researchers have used vector processors [18], SIMD [15] machines, and Hypercube (MIMD) [25] machines.

Another do-smarter approach is *weak mutation* [10]. Rather than running the entire program to completion and then checking the output, weak mutation halts the execution immediately after the mutated portion of the program is executed. The intermediate state is then examined, and the mutant is killed if the state differs from the expected state. While the intermediate state may not propagate to the end of execution, ergo "weak," experiments have shown that weak mutation tests are almost as effective as strong mutation tests but save 50% or more on computation cost [22].

As mentioned, do-faster approaches try to generate and run mutants more quickly. Most early mutation systems used interpretation, so mutants are executed slower than if compiled. The simplest compilation approach is separate compilation, which individually compiles, links, and runs each mutant. These mutation systems are easier to develop and the mutants are executed faster, however compiling each mutant separately is very slow and they require a great deal of storage.

Compiler-integrated mutation [6] mutates linked object code, thereby obviating the need for compiling mutants. Unfortunately, retrofitting this type of mutation system to a compiler is complex, time consuming, error-prone, and expensive.

The *mutant schema generation* (MSG) [29] is also designed to avoid the slow speed of interpreters. MSG embeds many mutants into each line of source code, so that one source file incorporates all mutants. The resulting file only needs to be compiled once. During execution, parameters are used to specify which mutant to run.

Untch carried out an experiment across four sufficient sets of mutation operators, including the sets proposed by Wong, Offutt, and Namin, and the single statement deletion operator [28]. Untch used regression analysis to show that SDL generates the fewest mutants, and it was best at predicting the mutation score of the given test suite. Using only the SDL operator is a do-fewer approach that we call *SDL-mutation*. Deng et al. [7] confirmed this result with an experimental study with muJava, showing that SDL generates relatively few mutants and leads to tests that are highly effective at killing other mutants.

Mresa and Bottaci [19] presented an empirical evaluation of Fortran operators, considering cost and effectiveness. Then they used several high efficiency (cost-effective) mutation operators to form three selective sets of mutation operators and evaluate their cost-effectiveness, by comparing them with two sets of randomly selected mutants.

The effectiveness of the mutation operators was defined in terms of mutation scores, which is the same as this paper. They also proposed a cost metric for mutation operators that considers the cost of generating tests and identifying equivalent mutants. They used a metric to measure the cost of test generation in terms of the number of mutants for a special case in which testers have an automatic test data generation tool (using Godzilla [21]) and testers only needed to generated about 3% of the tests by hand. Therefore, this metric is not applicable when testers generate all tests by hand. Their cost metric was based on both the number of mutants and the number of program states examined. The authors did this because most equivalent mutants required humans to investigate program states, and many mutants could be generated for the same statement. Therefore, after the first equivalent mutant on a statement was identified, other equivalent mutants were often quite easy to identify. The average number of mutants per statement for a selective or random set of mutation operators in Mresa and Bottaci's paper was between nine and 14. However, the number of mutants per statement of a single mutation operator (one-op mutation) in our paper is much lower: 0.04 to 0.5. Thus, we do not gain the benefit by separating the cost of investigating program states from identifying mutants and this cost metric is not applicable for our case.

Section V proposes metrics to normalize the experimental results, compute the cost, and calculate the cost-effectiveness for each mutation operator. In addition, our metric incorporates different weights for the cost of test generation and identifying equivalent mutants, making it more widely applicable.

The current paper makes several contributions. First, we evaluate SDL with a new tool in a new language, Proteum for C. Second, we evaluate cost and effectiveness of each mutation operator, and third, compare them with random sampling.

III. The Statement Deletion Operator for C

The SDL operator has appeared in several mutation tools for several different languages. This study uses Proteum's (**Pro**gram **Te**sting **U**sing **M**utants) operators [3]. They follow, as closely as possible, the original C operators defined by Agrawal et al. [1]. Proteum's operators are divided into four classes: statement, operator, variable, and constant. In Proteum, following the original C operators proposition, SDL is named SSDL.[1]

SSDL in Proteum has some differences from SDL for Java in muJava [7]. It systematically removes each statement as well as all inner statements. In contrast with SDL for Java, SSDL does not change the conditions of selection or loop statements. It also does not change declarations, even when declarations include initialization assignments. The eleven mutants generated for the program in Figure 1(a) are shown in Figure 1(b).

An advantage of SDL-mutation is the low number of mutants, which is proportional to the number of statements in the program. Intuitively, one might think there should be no equivalent SDL mutants unless the statement is unreachable or otherwise useless. However Deng et al. [7] found a few situations when SDL mutants might be equivalent in Java.

C SSDL mutants could also be equivalent because C compilers are more permissive than Java compilers, allowing some unusual situations. Consider the example in Figure 2.

The first example is the mutant that deletes the initialization of variable i at line 5. 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 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 are normally considered to be equivalent, since no test is guaranteed to kill them.

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

12 void test_while()
13 {
14    int a, b, c, t;
15
16    while (a < 5)
17    {
18        t = t + b + c;
19        a++;
20    }
21 }
```

(a) Original program

	Lines removed
M1:	7
M2:	6, 7, 8
M3:	5, 6, 7, 8
M4:	10
M5:	9, 10
M6:	3, 4, 5, 6, 7, 8, 9, 10
M7:	18
M8:	19
M9:	17, 18, 19, 20
M10:	16, 17, 18, 19, 20
M11:	14, 15, 16, 17, 18, 19, 20

(b) SSDL mutants

Fig. 1. Examples of SSDL generated mutants.

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 could be 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 ignore mutants of this nature if the tests do not kill them.

[1] For convenience, we use "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.

```
int foo (int j, int k)
{
    int i;

    i = 0;
    // do something
    i = k * j;
    return i;
}
```

Fig. 2. Equivalent SSDL mutants.

IV. Effectiveness of Individual C Operators

The experiment in this paper is divided into two parts. We first reproduce the analysis of effectiveness and cost using SDL as a single operator. Then we ask the general question, how effective is each mutation operator by itself?

Our experiment attempts to develop an overview of the effectiveness and cost of each mutation operator. We first design, collect, and refine a set of mutation-adequate test cases for each mutation operator. This section describes the experiment, including the tool used, steps followed, and results.

A. Experimental Tool

We used Proteum for four reasons. First, it is one of the most advanced tools for mutation analysis, including features such as generating mutants for a given program, automatically executing test cases against mutants, and calculating and showing results such as mutation scores. Second, it has been used for teaching and research activities for more than 15 years. Third, it implements 75 mutation operators for C, providing us with a fundamental environment to evaluate and search an appropriate set of mutation operators in practice. Fourth, the SSDL operator has already been defined and implemented in Proteum.

B. Experimental Design

This study collects mutation-adequate test sets for each mutation operator, and then computes their effectiveness in terms of how many total mutants they can kill. We define the following research questions:

RQ1: What is the cost of applying each individual C language mutation operator?

RQ2: How effective are the test sets created using each C language mutation operator?

This study uses six steps:

1) **Subjects:** We chose 39 C programs as experimental subjects, $S = \{s_1, s_2, \ldots, s_{39}\}$. These programs are described in the next section.
2) **Generating mutants:** We used Proteum to generate all the mutants $M = \{m_1, m_2, \ldots, m_{39}\}$, where m_i is the set of mutants for subject s_i. Because Proteum has 75 mutation operators, each m_i consists of 75 subsets of mutants $m_i = m_{i_{op_1}} \cup m_{i_{op_2}} \cup \ldots \cup m_{i_{op_{75}}}$. Each $m_{i_{op_j}}$ corresponds to the mutants of operator j for program s_i.
3) **Collecting the test pool:** We used a "universe" [8] of tests for each subject s_i by manually designing a pool of tests $T = \{t_1, t_2, \ldots, t_{39}\}$, where t_i kills all mutants of subject s_i (mutation adequate).
4) **Identifying equivalent mutants:** Equivalent mutants were identified by hand while designing adequate test sets. They are labeled $EQ = \{eq_1, eq_2, \ldots, eq_{39}\}$.
5) **Finding adequate subsets:** For each set of mutants from one subject and one operator, $m_{i_{op_j}}$, we ran tests within t_i until all mutants were killed. This resulted in a subset of tests, $t_{i_{op_j}} \subseteq t_i$, that was mutation-adequate for all mutants of $m_{i_{op_j}}$.
6) **Calculating mutation scores:** The mutation score of a test set directly indicates its effectiveness. To evaluate the effectiveness of each mutation operator, we ran each mutation-adequate set of tests $t_{i_{op_j}}$ against all mutants m_i, calculating their mutation scores as $MS_{i_{op_j}}$.

$$MS_{i_{op_j}} = MS(t_{i_{op_j}}, m_i) \quad i = 1, \ldots, 39, j = 1, \ldots, 75$$

where $MS(t, m)$ is the mutation score of test set t against the set of mutants m.

In steps 3 and 4, test cases were selected to kill all non-equivalent mutants by hand. Some programs had test sets from previous studies, and we added new tests by hand until all mutants were killed. For programs with no tests, we used the usual strategy in testing research, analyzing the mutants and generating tests by hand. When used in the order they were introduced, all the test cases are necessary, in the sense that all of them kill at least one mutant. Using previous terminology, our test sets are effective but some may be redundant [19].

In step 5 test sets for each mutation operator were selected from the adequate test set. We randomly picked tests from the adequate test set, discarding tests that did not kill new mutants, until a subset was constructed that killed all mutants of that operator (adequate for that operator). Again, these test sets are effective and may be redundant. To avoid possible bias caused by the selection process, we repeated this process 10 times, each time with a different random selection of test cases. So, the mutation score and the number of test cases for each operator, computed in step 6, are the average of the 10 different test sets adequate for each mutation operator.

C. Subjects

We chose 39 C programs of varying sizes and domains as experimental subjects. The subject programs vary in size from one to 20 functions, and from seven to 394 lines of code, totaling 189 functions and 2808 lines of code. Program mutation is primarily used for unit testing, so we focused on program units (C functions) rather than large systems. Table I summarizes the subject programs used in this experiment, showing the number of functions and total lines of code for each program.

D. SDL-mutation Results

This study replicates the Java SDL-mutation study [7] with four differences. (1) This study includes additional operators.

TABLE I
EXPERIMENTAL RESULTS FOR SSDL-MUTATION.

			SSDL Mutants			All Proteum Mutants				
Program	**Functions**	**LOCs**	**Mutants**	**Equiv.**	**Tests**	**Mutants**	**Equiv.**	**Tests**	**Killed**	**MS**
boundedQueue	6	49	45	4	5	1053	95	13	890.94	0.93
cal	1	18	11	0	3.5	845	67	8	723.54	0.93
Calculation	7	46	36	4	6.3	1059	101	13	881.36	0.92
checkIt	1	9	6	0	2	97	3	9	61.1	0.65
CheckPalindrome	1	10	8	0	3.1	158	18	8	130.2	0.93
countPositive	1	9	7	0	1.6	143	9	5	119.26	0.89
date-plus	3	132	89	1	15.2	2329	151	44	2090.88	0.96
DigitReverser	1	17	15	0	1.3	462	39	5	410.31	0.97
findLast	1	10	7	0	2	185	17	6	132.72	0.79
findVal	1	7	5	0	1.7	182	18	7	144.32	0.88
Gaussian	6	23	24	0	4.3	1009	19	21	950.4	0.96
Heap	7	41	38	2	3.4	1003	92	8	883.67	0.97
InversePermutation	1	15	13	0	3.3	551	61	12	450.8	0.92
jday-jdate	2	49	32	1	4.3	2660	75	27	2481.6	0.96
lastZero	1	9	7	0	1	165	9	5	134.16	0.86
LRS	5	51	37	3	3.1	1075	243	8	807.04	0.97
MergeSort	3	32	23	0	3.3	937	46	18	846.45	0.95
numZero	1	10	7	0	1.3	143	17	5	117.18	0.93
oddOrPos	1	9	7	0	1.9	335	63	7	223.04	0.82
pcal	8	204	136	3	15.2	6109	740	49	5046.86	0.94
power	1	11	9	0	2.5	256	12	9	234.24	0.96
print_tokens	17	349	227	26	10.1	4172	530	34	3605.58	0.99
print_tokens2	18	275	232	26	6.5	4482	626	27	3778.88	0.98
printPrimes	2	35	25	1	2.1	686	62	7	611.52	0.98
Queue	6	64	51	0	7.6	437	25	12	407.88	0.99
quicksort	1	23	22	1	2.2	992	82	13	873.6	0.96
RecursiveSort	1	17	14	1	1.4	535	43	8	462.48	0.94
replace	20	394	256	30	19.6	10,617	2000	143	8358.49	0.97
schedule	18	213	148	17	10.5	2026	211	45	1760.55	0.97
schedule2	16	195	143	24	10.9	2515	396	41	2034.24	0.96
Stack	6	56	49	9	5.1	435	42	11	389.07	0.99
stats	1	19	17	2	1.8	843	97	7	701.24	0.94
sum	1	7	5	0	1.5	157	11	6	124.1	0.85
tcas	8	63	46	4	11.5	2285	404	62	1636.47	0.87
testPad	1	24	16	1	3.7	596	55	14	503.13	0.93
totInfo	7	214	126	9	10.6	6326	653	49	5446.08	0.96
trashAndTakeOut	2	19	15	2	3.6	564	26	12	484.2	0.90
twoPred	1	10	7	1	2	232	24	10	162.24	0.78
UnixCal	4	119	101	4	7.3	4619	336	27	4154.51	0.97
Total	189	2857	2062	176	203.3	63,275	7518	815	53,254.33	0.96
Min	1	7	5	0	1	97	3	5	61.1	0.65
1st qu.	1	10.5	8.5	0	2	295.5	21.5	7.5	228.64	0.91
Mean	4.85	73.26	52.87	4.51	5.21	1622.44	192.77	20.90	1365.50	0.92
Trim. Mean	3.61	51.39	37.94	2.26	4.41	1131.10	111.84	15.87	962.15	0.94
3rd qu.	6.5	63.5	50	4	6.9	2155.5	181	27	1698.51	0.97
Max	20	394	256	30	19.6	10,617	2000	143	8358.49	0.99

(2) The subjects are in C, not Java. (3) The tool used was Proteum, not muJava, and the SSDL operator is implemented differently. (4) Different subject programs were used.

Table I shows the data from the study. The columns under SSDL show the results from the SSDL-adequate tests. For example, *boundedQueue* had 45 SSDL mutants, four were equivalent, and five tests were needed to kill the rest (on average, since this number is the average size of 10 test sets). The columns under Proteum show data from running adequate tests on all mutants. For example, Proteum generated 1053 mutants for *boundedQueue*, 95 were equivalent, and 13 tests were needed to kill all mutants. The SSDL-adequate tests killed 93% of all non-equivalent mutants. The "Killed" column shows how many total mutants were killed by the SSDL test sets. This value is the average over the 10 test sets and is computed by $MS * (Mutants - Equiv)$. The "Equiv." column shows the number of equivalent mutants. The first author determined these by hand over a period of several months.

The 39 subjects had a total of 2062 mutants from the SSDL operator, and 63,275 from all operators. This is a percentage increase of 2968%. The number of SSDL mutants ranged from five (in *findVal* and *sum*) to 256 (in *replace*). The total number of mutants ranged from 97 (in *checkIt*) to 10,617 (in *replace*). As can be seen in Table I, the program *replace* had the most total and SSDL mutants, increasing 4047% from the SSDL mutants to all mutants.

Given that the size of the programs varies significantly, we used the *trimmed mean* (at 25%) because it is less sensitive to outliers than the mean. Also, the trimmed mean does not require the subjective removal of outliers. According to the results, SSDL generated an average of 37.94 mutants while all operators generated an average of 1131.10 mutants.

Table I shows an advantage of SSDL: it generates few equivalent mutants. 176 of the 2062 SSDL mutants were found equivalent (8.54%), while 7518 out of all 63,275 mutants were found equivalent (11.88%). On average (trimmed mean), SSDL generated approximately 2.26 equivalent mutants per program and the complete set of operators generated 111.84. For both SSDL and the complete set of operators, the program with the most equivalent mutants was *replace* (30 and 2000).

In terms of the number of tests, SSDL required only 203.3 tests, 24.94% of the number of tests needed to kill all other mutants (815). Again, *replace* required the most tests, 19.6 for the SSDL mutants and 143 for all mutants (a 629.59% increase). *lastZero* had the fewest tests, one for the SSDL mutants and five for all mutants (*countPositive* and *numZero* also only needed five tests).

These results indicate that SSDL-mutation is a cost-effective alternative to using the complete set of operators. Although the mean mutation score of .92 (.94 for the trimmed mean) may suggest that SSDL is not as strong, 76.92% (30) of the mutation scores in Table I are greater than .92. In addition, considering the total number of mutants killed, SSDL test cases were able to kill approximately 96% of all mutants. The lowest mutation score was .65 (*checkIt*) and four programs had a mutation score of .99. However, as shown in Figure 3, the scores for checkIt, twoPred (.78), findLast (.79), and oodOrPos (.82) are outliers. If we discard these outliers, the average mutation score is .94. A .90 mutation score is commonly regarded as difficult to obtain [9, 27], indicating that SSDL is a viable alternative to using all mutation operators, although not necessarily better.

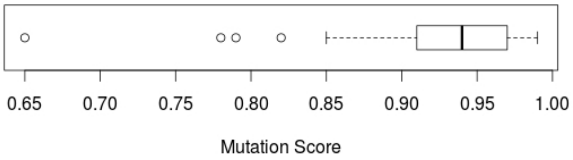

Fig. 3. Boxplot of the mutation scores obtained by executing the SSDL-adequate test set against the complete set of mutants.

E. One-op Mutation Results

The results in the last section confirm previous results for SDL-mutation in Java [7], showing that SSDL can reduce cost while still resulting in effective tests.

Our choice to investigate SSDL as a single mutation operator is based on the following analysis: it is guaranteed that every program will generate SDL mutants, the number of SDL mutants is bound by the LOC in the program, and the operator tends to generate few equivalent mutants according to previous studies. One thing missing from the previous studies is a similar empirical analysis with other operators. We generalize SDL-mutation to *one-op mutation*, where only one mutation operator is used, and ask whether other operators could be equally effective by themselves.

This section provides results similar to those in section IV-D, but for each individual operator Op_i. We use the same universe of tests and find 10 subsets of tests that are adequate for each operator Op_i, and then measure their mutation scores against all mutants, as was done with SSDL. The test sets are reduced so that every test kills at least one Op_i mutant.

Proteum has 75 operators, 52 of which created mutants for the programs, as shown in Table II.[2] Some C operators did not generate mutants for any of our subjects because they mutate language features that are not commonly used. Operators are defined by their four-character acronyms, as described by Agrawal et al. [1]. The mutation score (MS) column shows the average mutation score ($MS_{i_{op_j}}$) across the 39 subject programs. For example, the tests that killed all Cccr mutants killed 92% of all mutants across the 39 subjects. The Test Cases column shows the average percentage of all adequate tests needed to kill all non-equivalent mutants of that operator. For instance, Cccr adequate test sets are, on average, 31.79% the size of the test set that killed all mutants. The Mutants column shows the percent of all mutants that are created from this operator. The Equiv Mutants column shows the percentage of mutants from that operator that are equivalent. So, 31.79% of the complete universe of tests were needed to kill all the Cccr mutants, 4.98% of all mutants were Cccr, and 10.30% of Cccr mutants were equivalent. We also included sets of 5%, 10%, 15%, and 20% randomly sampled mutants at the bottom for comparison. These mutants were selected by choosing $X\%$ of mutants from each operator, thus the sets were not always exactly $X\%$ of the total.

Most operators had high mutation scores (only nine were below 70%, which is considered a very low mutation score), several were as high or higher than SSDL's score. Table III shows the 16 highest mutation scores, using 90% as the cutoff.

Although the overall mutation score is certainly important, it is not the only important criterion for choosing an operator for one-op mutation. The mutation score is a good measure for effectiveness, but we also want to decrease cost, which is more complicated. The number of mutants contributes to the cost, as each mutant must be executed until it is killed. The number of tests needed also contributes to cost, as each test has to be stored, run, and checked, possibly many times. The number of equivalent mutants also contributes to cost, as they need to be analyzed separately, often by hand, and sometimes with great difficulty.

To choose the "best" one-op mutation operator, we perform a cost-effectiveness analysis (CEA) by aggregating cost and effectiveness into one formula. We focus primarily on human costs, as computer costs are orders of magnitude lower, and can be reduced by using faster computers.

The number of mutants directly affects only computer costs as the cost is in storage and execution. The number of test cases and the number of equivalent mutants, on the other hand, also affect human effort. The tester must design and construct the tests, then run each test potentially many times, each time evaluating the results. The tester also must identify equivalent mutants, usually by a difficult manual analysis. Therefore we base our cost estimation on the number of tests and equivalent mutants, and not the number of mutants.

Ideally, we would like a mutation operator that maximizes the mutation score, while minimizing the tests and equivalent mutants. Clearly, no single operator satisfies all three goals.

[2] Operators not used in the experiment are listed in Table V.

TABLE II
OPERATOR SUMMARY.

Operator	MS	Test Cases	Mutants	Equiv.
Cccr	0.92	31.79%	4.98%	10.30%
Ccsr	0.95	36.75%	6.90%	2.57%
CRCR	0.96	40.58%	14.00%	3.07%
OAAA	0.76	11.66%	0.74%	0.62%
OAAN	0.85	20.47%	2.34%	4.89%
OABA	0.74	10.58%	0.45%	0.95%
OABN	0.84	22.66%	1.61%	7.98%
OAEA	0.79	11.23%	0.20%	0.00%
OALN	0.82	16.74%	1.27%	0.95%
OARN	0.83	19.24%	3.81%	1.72%
OASA	0.74	9.83%	0.30%	0.00%
OASN	0.83	18.68%	1.07%	5.03%
OBAN	0.73	5.88%	0.45%	1.90%
OBBN	0.69	4.57%	0.18%	2.38%
OBLN	0.72	5.39%	0.18%	2.38%
OBNG	0.72	6.00%	0.27%	0.00%
OBRN	0.84	9.72%	0.54%	0.79%
OBSN	0.73	6.28%	0.18%	0.00%
OCNG	0.79	17.65%	0.83%	0.57%
OCOR	0.26	2.04%	0.57%	93.49%
OEAA	0.88	25.59%	4.06%	17.44%
OEBA	0.82	22.40%	2.28%	33.59%
OESA	0.80	19.04%	1.52%	12.13%
Oido	0.78	13.49%	0.53%	0.57%
OIPM	0.55	3.70%	0.04%	0.00%
OLAN	0.78	16.59%	1.68%	25.15%
OLBN	0.73	12.90%	1.00%	41.58%
OLLN	0.67	10.84%	0.34%	2.05%
OLNG	0.77	12.43%	1.03%	0.59%
OLRN	0.85	23.50%	2.06%	10.78%
OLSN	0.76	14.42%	0.67%	3.30%
ORAN	0.92	31.79%	3.77%	11.73%
ORBN	0.91	28.33%	2.25%	15.64%
ORLN	0.89	24.40%	1.66%	8.64%
ORRN	0.95	39.53%	4.15%	14.33%
ORSN	0.87	22.90%	1.50%	11.39%
SBRC	0.45	3.60%	0.09%	41.67%
SBRn	0.87	10.71%	0.16%	0.00%
SCRB	0.69	3.47%	0.03%	0.00%
SGLR	0.62	2.04%	0.16%	80.00%
SMTC	0.80	13.56%	0.34%	2.66%
SMTT	0.76	12.31%	0.34%	0.00%
SMVB	0.66	10.40%	0.23%	8.00%
SRSR	0.89	24.87%	4.50%	3.83%
SSDL	0.92	28.80%	3.81%	5.11%
SSWM	0.83	10.15%	0.45%	2.94%
STRI	0.87	23.22%	1.12%	1.17%
STRP	0.88	25.02%	3.81%	0.69%
SWDD	0.36	6.73%	0.20%	51.46%
VDTR	0.96	53.09%	8.18%	31.33%
VGAR	0.84	16.87%	1.21%	8.50%
VGPR	0.85	11.96%	1.18%	6.11%
VGSR	0.96	32.55%	9.96%	5.49%
VLAR	0.79	8.61%	0.36%	11.57%
VLPR	0.80	12.47%	1.01%	1.13%
VLSR	0.95	38.72%	11.82%	4.68%
VSCR	0.74	18.63%	2.60%	2.22%
VTWD	0.95	42.55%	5.45%	8.46%
Varr	0.83	14.55%	1.17%	12.67%
Vprr	0.82	14.06%	1.27%	1.50%
Vsrr	0.96	40.39%	13.35%	4.55%
Random 5%	0.96	39.92%	5.12%	9.49%
Random 10%	0.98	49.48%	10.12%	8.70%
Random 15%	0.99	54.66%	14.38%	9.62%
Random 20%	0.99	59.52%	20.11%	8.93%

Table III shows that Ccsr and CRCR have the highest mutation scores, ORBN has the fewest test cases, and Ccsr has the fewest equivalent mutants.

To account for all three factors, we have designed a weighted cost function, which is then divided by the mutation score. The weighted cost is based on relative cost rather than absolute cost, by hypothesizing a perfect operator whose tests kill all mutants (MS is 100%), has no equivalent mutants (%Equiv is 0%), and uses a minimum, but positive, number of tests. We calculate the cost of the other mutation operators by comparing them with this hypothetical perfect operator.

The ranges for %Test Cases and %Equiv Mutants are different, so we normalize them. We first subtract the smallest value (2.04% for the tests and 0.0% for the equivalent mutants) then divide by the difference between the smallest and largest values (60.15% for the tests and 93.49% for the equivalent mutants). The normalization formula is:

$$Normalized\ Data = \frac{data - MINvalue}{MAXvalue - MINvalue} \tag{1}$$

The normalized numbers are in the %Norm Test Cases and %Norm Equiv Mutants columns in Table III. The cost column is the sum of the two normalized numbers, and represents the cost of using just the one operator. The cost function of a mutation operator is formally defined as:

$$\begin{aligned} Cost(OP) = \%Norm\ Test\ Cases(OP) \times W_t + \\ \%Norm\ Equiv\ Mutants(OP) \times W_e \end{aligned} \tag{2}$$

where W_t and W_e are constant weighting factors. Setting W_t and W_e to 1 assumes the cost of generating tests and determining equivalent mutants is the same. This assumption is used in Table III. However, if an organization has a very good automatic test generator, then W_t might be smaller. Likewise, if an automatic equivalent mutant detector is used, then W_e might be smaller. This definition assumes that the number of test cases and the number of equivalent mutants are independent.

Finally, the cost-effectiveness measure to evaluate the best mutation operator, MOCEA (Mutation Operator Cost Effectiveness Analysis) is given by formula 3. As is usual with cost-effective formulas, low values are more cost-effective.

$$\text{MOCEA}(OP) = \frac{Cost(OP)}{MS(OP)} \tag{3}$$

The normalization procedure we used is common, but of course could be done differently. In fact, we tried several alternatives, and they all resulted in the same cost-effectiveness ordering among the mutation operators. The free variables are the weights, so we also looked into how different weights would affect the results. Table IV presents several possible combinations of values for the weights W_t and W_e in the computation of MOCEA according to the data collected in the present experiment. The best operator value is marked in bold, and SSDL is the most cost-effective for three of the five combinations. If test generation is free ($W_t = 0$), the CRCR or Ccsr operators become more cost-effective, and if equivalent detection is irrelevant ($W_e = 0$), the ORBN operator becomes more cost-effective.

TABLE III
OPERATORS WITH MUTATION SCORES OVER .90.

Operator	**MS**	**%Test Cases**	**%Norm Test Cases**	**%Mutants**	**%Equiv Mutants**	**%Norm Equiv Mutants**	**Cost**	**MOCEA**
Random 20%	0.99	59.52	100.00	20.11	8.93	9.55	109.55	1.11
Random 15%	0.99	54.66	91.54	14.38	9.62	10.29	101.83	1.03
Random 10%	0.98	49.48	82.53	10.12	8.70	9.31	91.84	0.94
VDTR	0.96	53.09	88.81	8.18	31.33	33.51	122.33	1.27
CRCR	0.96	40.58	67.05	14.00	3.07	3.28	70.33	0.73
Random 5%	0.96	39.92	65.90	5.12	9.49	10.15	76.05	0.79
VGSR	0.96	32.55	53.08	9.96	5.49	5.87	58.95	0.61
Vsrr	0.95	49.31	82.24	13.81	4.83	5.17	87.40	0.92
VTWD	0.95	42.55	70.48	5.45	8.46	9.05	79.53	0.84
ORRN	0.95	39.53	65.22	4.15	14.33	15.33	80.55	0.85
VLSR	0.95	38.72	63.81	11.82	4.68	5.01	68.82	0.72
Ccsr	0.95	36.75	60.39	6.90	2.57	2.57	63.14	0.66
ORAN	0.92	31.79	51.76	3.77	11.73	12.55	64.30	0.70
Cccr	0.92	31.79	51.76	4.98	10.30	11.02	62.77	0.68
SSDL	0.92	28.80	46.56	3.81	5.11	5.47	52.02	0.57
ORBN	0.91	28.33	45.74	2.25	15.64	16.73	62.47	0.69

TABLE IV
DIFFERENT WEIGHTS FOR COMPUTING MOCEA

Operator	$W_t = 1$ $W_e = 1$	$W_t = .5$ $W_e = 1$	$W_t = 0$ $W_e = 1$	$W_t = 1$ $W_e = .5$	$W_t = 1$ $W_e = 0$
Random 20%	1.11	0.60	0.10	1.06	1.01
Random 15%	1.03	0.57	0.10	0.98	0.92
Random 10%	0.94	0.52	0.09	0.89	0.84
VDTR	1.27	0.81	0.35	1.10	0.93
CRCR	0.73	0.38	**0.03**	0.72	0.70
Random 5%	0.79	0.45	0.11	0.74	0.69
VGSR	0.61	0.34	0.06	0.58	0.55
Vsrr	0.92	0.49	0.05	0.89	0.87
VTWD	0.84	0.47	0.10	0.79	0.74
ORRN	0.85	0.50	0.16	0.77	0.69
VLSR	0.72	0.39	0.05	0.70	0.67
Ccsr	0.66	0.35	**0.03**	0.65	0.64
ORAN	0.70	0.42	0.14	0.63	0.56
Cccr	0.68	0.40	0.12	0.62	0.56
SSDL	**0.57**	**0.31**	0.06	**0.54**	0.51
ORBN	0.69	0.44	0.18	0.59	**0.50**
Minimum	0.57	0.31	0.03	0.54	0.50

V. DISCUSSION

This paper introduces the concept of *one-op mutation analysis*, in which only one mutation operator is used as a reasonably effective, less expensive alternative. This is an example of a do-fewer approach. Previous papers [7, 28] have suggested using only the statement deletion operator (SDL), so we began by investigating it.

Deng et al. [7] studied SDL in Java programs using the muJava system. Section IV-D presents similar results but uses C, the Proteum tool, and a somewhat different version of the SDL operator (SSDL in Proteum). Table I shows that the number of SSDL mutants is only 3.26% of all mutants. muJava's SDL mutants were 18.8% of all mutants, primarily because muJava applies a selective strategy [26] and thus has fewer mutation operators. The tests that killed all SSDL mutants killed on average 92% of all Proteum's mutants, which is the same number killed for muJava's mutants.

The weakest results were on very small programs. The SSDL tests had mutation scores less than 80% on three programs that had 10 lines or less (*checkIt*, *twoPred* and *findLast*). We found no general correlation between mutation score and program size, but since this effect was only on very small programs, the cost of using more mutants, or even all mutants, is relatively small.

One of the most important advantages of the SSDL operator is that it generates relatively few equivalent mutants. Only 8,5% of the SSDL mutants were equivalent, compared with 11.9% of all mutants. This matches intuition – every statement should contribute something to the program, so if removing it does not change the program's behavior, why is it there? Some SSDL mutants are equivalent because of idiosyncrasies of C, and others because of an engineering practice of including redundancy to add strength and stability.

A related advantage is that many equivalent SSDL mutants can be detected automatically. This is generally an undecidable problem [23], but static analysis can detect many equivalent mutants, and, based on our observation, is easier for many SSDL mutants than for most other types of mutants.

The second empirical result is a study to evaluate other mutation operators as a candidate for one-op mutation. We collected data for each C operator in Proteum and computed a mutation score for each operator. We then introduced a cost-effectiveness analysis metric, MOCEA, to use a multi-dimensional comparison of the operators in terms of cost and effectiveness. We concluded that the SSDL operator is the most cost-effective one-op operator if the two factors associated with the cost are equally balanced. In other configurations, other operators may perform better, as, for instance if a very good automatic test data generator is used so the cost to generate test cases can be neglected.

Another advantage of the SSDL operator, one that is not captured in the cost-effectiveness function, is that all programs will contain SSDL mutants. This is not true, for instance, for Ccsr, another of the more cost-effective operators. Such an operator should be avoided since it is not applicable in some cases. Table V shows how many of our 39 subject programs had at least one mutant of each type.

Our metric can be tailored to local conditions by adjusting the weight values (W_t and W_e). It does not take into account all factors, for example, the variance in scores or the applicability of the operator for all programs.

TABLE V
NUMBER OF PROGRAMS IN WHICH EACH OPERATOR APPEARS

Operators present in ...	
all programs	**none of the programs**
CRCR, OCNG, SSDL, SRSR, STRP, VDTR, Vsrr, VTWD,	OBAA, OBBA, OBEA, OBSA, OSAA, OSAN, OSBA, OSBN, OSEA, OSLN, OSRN, OSSA, OSSN, SCRn, SDWD, Vtrr
some programs	
Ccsr (39), ORLN (39), ORRN (39), ORAN (39), OEAA (38), OEBA (38), OESA (38), ORBN (38), ORSN (38), STRI (37), Oido (35), SMTC (35), SMTT (35), Cccr (31), OAAN (31), OALN (30), OARN (30), OABN (29), OASN (29), SMVB (25), OLAN (20), OLBN (20), OLLN (20), OLNG (20), OLRN (20), OLSN (20), SWDD (16), Varr (9), Vprr (9), OAAA (8),OAEA (8), VSCR (8), OABA (7), OASA (7), SBRC(7), SSWM (7), OCOR (5), OBAN (3), OBBN (3), OBLN(3), OBNG (3), OBRN (3), OBSN (3), SBRn (3), OIPM (1), SCRB (1), SGLR (1)	

A. Threats to Validity

The threats to validity in this paper are common in software engineering experiments. As is usually the case, we cannot be sure the subject programs are representative. We mitigated this problem by selecting programs from various sources, various domains, and to be of different sizes.

The process of building an adequate test set for each subject program (the universe of test cases) was tedious and time consuming. This made building more than one test set per program prohibitively expensive. This may represent a threat but 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.

When selecting test sets for each mutation operator, the complete universe of tests was scanned, in order, until an adequate test set was obtained. To reduce the effect of the order of the individual tests, we randomly selected 10 different test sets.

Identifying equivalent mutants in a large experiment like this is also error-prone. This was done manually by the first author. Effort data was not collected, but it was done over a period of months. Some non-equivalent mutants may have been incorrectly assessed as equivalent. This error is probably at the noise level.

The MOCEA cost-effectiveness metric is a new way to evaluate mutation operators. It poses a construct validity threat and more extensive use is required to validate and tune it.

VI. CONCLUSIONS AND FUTURE WORK

This paper generalizes the concept of SDL-mutation [7] to *one-op* mutation, and presents substantial experimental results of the concept on C programs. It introduces a novel cost-effectiveness measure (CEA) for one-op mutation and evaluates the measure on 39 C programs with all 75 Proteum C mutation operators. The CEA uses the average mutation score of tests that kill all mutants for the single operator being measured as a measure of approximation, and incorporates two elements of cost, the number of tests needed and the number of equivalent mutants.

The empirical data show that the statement deletion operator (SSDL in C) is the most cost-effective single mutation operator if the number of tests and the number of equivalent mutants are weighted equally. However, if test generation is given a zero weight, CRCR and Ccsr become more cost-effective. This can be a valuable tradeoff for practicing testers who might have different tools or expertise on the testing team. Future mutation systems for C should allow testers to choose one-op mutation, and define it to be either the SSDL, the CRCR, or the Ccsr operator. Despite the differences in C and Java and the differences in how Proteum and muJava implement SDL, the results presented here are very consistent with the evaluation of SDL for Java. We are not able to definitively evaluate how these differences affect our results.

In the future, we want to consider the idea of *two-op mutation*, where two operators are used to complement each other to be more cost-effective than either by themselves. The operators that modify arithmetic or logical operators seem to be likely candidates.

In section V, we observed that some operators did not create any mutants of our programs, and the tester would not need to create any tests. The cost-effective formula does not measure this effect, but if it did, would rank SSDL even higher for one-op. It would also be useful to compare one-op mutation with other techniques, such as selective mutation and random sampling.

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. He also thanks Ms. Lisa Campos for her support during the conduct of this work.

REFERENCES

[1] 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. Technical report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette IN, March 1989.

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

[3] 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)*, pp. 79–95, New Brunswick, NJ, July 1996.

[4] R. A. DeMillo and J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[5] 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.

[6] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proceedings*

of the Fifteenth Annual Computer Software and Applications Conference (COMPSAC' 92), Tokyo, Japan, September 1991. Kogakuin University, IEEE Computer Society Press.

[7] 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.

[8] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software, Elsevier*, 38 (3):235–253, 1997.

[9] M. J. Harrold, J. Offutt, and K. Tewary. An approach to fault modeling and fault seeding using the program dependence graph. *Journal of Systems and Software, Elsevier*, 36(3):273–296, March 1997.

[10] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[11] 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.

[12] G. Kaminski, P. Ammann, and J. Offutt. Improving logic-based testing. *Journal of Systems and Software, Elsevier*, 2012. To appear.

[13] 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*, pp. 4–100, San Jose, CA, October 2000. Wiley's Software Testing, Verification, and Reliability, December 2001.

[14] K. N. King and J. Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.

[15] E. W. Krauser, A. P. Mathur, and V. Rego. High performance testing on SIMD machines. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 171–177, Banff, Alberta, July 1988. IEEE Computer Society Press.

[16] N. Li, U. Praphamontripong, and J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Fifth Workshop on Mutation Analysis (IEEE Mutation 2009)*, Denver CO, April 2009.

[17] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava : An automated class mutation system. *Software Testing, Verification, and Reliability, Wiley*, 15(2):97–133, June 2005.

[18] A. P. Mathur and E. W. Krauser. Modeling mutation on a vector processor. In *10th International Conference on Software Engineering*, pp. 154–161, Singapore, April 1988. IEEE Computer Society Press.

[19] E. S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification, and Reliability, Wiley*, 9(4):205–232, 1999. December.

[20] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, pp. 351–360, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368136.

[21] J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[22] J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[23] J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification, and Reliability, Wiley*, 7(3):165–192, September 1997.

[24] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pp. 45–55, San Jose, CA, October 2000.

[25] J. Offutt, R. Pargas, S. V. Fichter, and P. Khambekar. Mutation testing of software using a MIMD computer. In *1992 International Conference on Parallel Processing*, pp. II–257–266, Chicago, Illinois, August 1992.

[26] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[27] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: Deterministic versus random input generation. In *Fault-Tolerant Computing: The Twenty-First International Symposium*, pp. 410–417, Montreal, Canada, June 1991. IEEE Computer Society Press.

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

[29] R. Untch, J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pp. 139–148, Cambridge MA, June 1993.

[30] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993. (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

[31] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software, Elsevier*, 31(3):185–196, December 1995.

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