# Empirical Evaluation of the Statement Deletion Mutation Operator

Lin Deng, Jeff Offutt, Nan Li
Software Engineering
George Mason University
Fairfax, VA 22030, USA
{ldeng2, offutt, nli1}@gmu.edu

***Abstract*—Mutation analysis is widely considered to be an exceptionally effective criterion for designing tests. It is also widely considered to be expensive in terms of the number of test requirements and in the amount of execution needed to create a good test suite. This paper posits that simply deleting statements, implemented with the statement deletion (SDL) mutation operators in Mothra, is enough to get very good tests. A version of the SDL operator for Java was designed and implemented inside the muJava mutation system. The SDL operator was applied to 40 separate Java classes, tests were designed to kill the non-equivalent SDL mutants, and then run against all mutants.**

## I. INTRODUCTION

Program mutation analysis [1], [2] creates alternate versions of programs (*mutants*), each of which differs from the original by a small syntactic change, then asks the tester to design inputs to *kill* the mutants by causing each mutant to have a different result from the original version. These changes are defined by *mutation operators*, which are rules that specify changes to syntactic elements in a program. The ability for mutation testing to help testers design high quality tests has always depended directly on the mutation operators. Well designed mutations operators can result in very powerful testing, but poorly designed operators can result in ineffective tests.

Mutation operators have been designed for several programming languages, including Fortran 77 [2], [3], C [4], and Java [5], [6]. Jia and Harman surveyed mutation analysis for programs and other software engineering artifacts [7]. The operators that modify individual statements (*statement-level operators*) have been fairly stable since the Mothra project [3], with the major change being from the *selective* operator study [8], where it was found that using five Mothra mutation operators for Fortran yielded tests that killed most other mutants. This approach was implemented in subsequent mutation systems, including muJava [6] with 15 operators for Java.

However, users of mutation have observed that mutation creates many test requirements (that is, mutants) relative to the number of test requirements from other test criteria. Budd [9], analyzed the number of mutants and found it to be roughly proportional to the product of the number of variable references times the number of data objects ($O(Refs * Vars)$). Acree et al. [10] later claimed that the number of mutants is $O(Lines * Refs)$–assuming that the number of data objects in a program is proportional to the number of lines. This was reduced to $O(Lines * Lines)$ for most programs; this figure appears in most of the literature.

Offutt et al. [8] performed a statistical regression analysis of actual programs, showing that the number of lines did **not** contribute to the number of mutants, but that Budd was correct. That paper also introduced *selective mutation*, which reduces the number of mutants to be proportional to the number of variable references ($O(Refs)$).

More recently, Li et al. found that although selective mutation has far more test requirements than the edge-pair, all-uses and prime path criteria, it ultimately needs **fewer** tests [11]. This implies that many mutants are redundant, encouraging us to believe that mutation testing can be effective with fewer mutants.

This paper presents results that investigate a hypothesis that we can reduce the overlap among mutants with a simple, direct approach: a single mutation operator. Untch [12] proposed this idea by using the *statement deletion operator* (SDL), and got positive initial results. SDL removes entire statements from the program, challenging the tester to design tests that cause each statement to have an effect on the outcome. Note this is considerably more from statement coverage, which only requires that a statement be reached, rather than omitting the statement cause an error. muJava implements statement coverage with the special "*Bomb()*" operator. The SDL operator does not mimic user errors, but does require very strong tests.

Section II describes similar research into reducing the cost of applying mutation. Section III then defines the SDL mutation operator in detail, including subtle decisions that have to be made to use it in Java. Section IV presents results from an empirical evaluation of this hypothesis, and section V presents conclusions and recommendations.

## II. BACKGROUND AND RELATED WORK

Statement deletion has been in mutation systems since the beginning, including the Mothra system [2]. DeMillo, Pan and Spafford [13] proposed a technique called *critical slicing*

that was based on SDL and mutation testing. The idea is to delete statements to help localize faults during debugging. The purpose of this research is to reduce the expense of mutation analysis for testing.

Several approaches have been developed to reduce this expense. Untch categorized the approaches into three strategies, *do-fewer*, *do-smarter*, and *do-faster* [14], [15]. Do-fewer approaches try to run fewer mutated programs without incurring intolerable loss in effectiveness. Do-smarter approaches distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs. Do-faster approaches try to generate and run mutant programs more quickly.

An early do-fewer approach [16] randomly sampled from all created mutants. This is appealing because it is simple and straightforward, but the resulting tests were significantly weaker when the sampling rate was made low enough to yield appreciable savings. Wong and Mathur suggested the idea of *selective mutation*, which uses only the most critical mutation operators [17], [18]. Offutt et al. [8] developed this idea with a detailed study and recommended reducing the mutation operators used by Mothra [2] from 22 to five. This selective set provided almost the same test coverage as non-selective mutation, and was used in muJava [6] with 15 operators for Java.

More recent work on selective mutation was by Namin et al. [19]. They viewed selective mutation statistically, and looked at it as a variable selection or reduction problem. They started with 108 C mutation operators and used statistical approaches to find 28 mutation operators that are sufficient to predict how effective a test suite could be on all operators. This reduced the number of mutants by 92%, the highest rate of reduction compared with other approaches. An interesting point of Namin's work is that instead of just looking at how close the selective set comes to 100% coverage, they looked at the entire range of possible scores. Tests were chosen randomly from a given pool, which could have affected the results.

Kaminski et al. investigated a do-fewer approach theoretically [20]. The *relational operator replacement* mutation operator (*ROR*) generates seven mutants for every relational operator in a program. Kaminski proved that if three mutants are killed for each relational operator, those tests are guaranteed to kill the remaining four mutants. Thus only three mutants need be created. Just et al. had similar results with the conditional operator replacement mutation operator [21].

Most do-smarter approaches have used non-standard computer architectures by distributing the computational expense of running hundreds of mutants over several machines. Because each execution is independent, this problem lends itself very well to parallelism. Mutation analysis has been adapted to vector processors [22], SIMD machines [23], Hypercube (MIMD) machines [24].

Another do-smarter approach is *weak mutation* [25]. Weak mutation halts execution shortly after the mutated portion of the program, rather running the program to completion. The intermediate state is then examined, and the mutant is killed if the state is incorrect. Although "weak" because the incorrect state would not always propagate to the end of execution, experiments showed that weak mutation tests were almost as effective as strong mutation tests, with a savings of 50% or more [26].

Do-faster approaches try to generate and run mutants more quickly. Most early mutation systems used interpretive execution, which is significantly slower than compiled speeds. The simplest approach is separate compilation, which individually creates, compiles, links, and runs each mutant. This avoids the interpretation cost, but creates a compilation bottleneck [27], making mutation even slower for most programs. Moreover separate compilation requires a great deal of storage. It is, however, very simple to build separate compilation mutation systems.

Compiler-integrated mutation [28] mutates linked object code, thus gaining compilation speed without a compilation bottleneck. However, crafting this type of system on top of an existing compiler is very difficult, time consuming, and expensive.

The *mutant schema generation* (MSG) [29] approach embeds many mutants into each line of source code, so that one source file contains all the mutants. This file is then compiled once, and run with an additional parameter to specify which mutant should be activated. The muJava system used a combination of MSG and Java reflection, which is similar to the compiler-integrated technique, but modifies Java bytecode rather than executable code.

Untch conducted an experiment across four sufficient sets of mutation operators, including the sets proposed by Wong, Offutt, and Namin, in addition to the single statement deletion operator (*SSDL* in his work, which was with C programs) [12]. Untch used regression analysis to show that SDL generates the fewest mutants, but is 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*. This paper evaluates this preliminary result in two different directions. First, we implement the SDL operator for Java, which involves a number of subtle decisions on the language constructs. Second, we evaluate its benefits in terms of how well tests generated to kill only SDL mutants perform when run on all of muJava's method-level mutants.

## III. The Statement Deletion Mutation Operator

The statement deletion operator was implemented in Mothra for Fortran-77, which has simple control structures, no statement blocks, no conditional returns, and no dynamic memory. SDL was also defined for C [30] and for Ada [31], both of which have more complicated control structure than Fortran-77. SDL requires much more than statement

coverage, because a killing test must cause the statement to affect the program's behavior. We define how SDL is applied to the various control structures in Java, then compare our Java SDL operator with the C and Ada versions.

We start by defining SDL on single statements, then extend the definition to other control structures. These extended rules follow four guidelines:

1) **All possible cases**: Every possible case must be considered.
2) **Boolean conditions**: Most control structures have at least one Boolean condition, which should be deleted.
3) **Inner statements**: Statements inside control structures must be deleted.
4) **Nested control structures**: Nested control structures must be treated recursively.

This paper provides enough documentation so that these results can be replicated in other tools and other experiments. To do so, we define SDL in terms of language constructs, and provide explicit examples for clarity.

**SDL for single statements.** Fortran has a CONTINUE statement, which only provides a placeholder. Thus, Mothra [3] implemented SDL by replacing each statement with CONTINUE. In Java, each statement is commented out, as shown in figure 1. SDL is not applied to variable declarations, since the mutants would not compile. This example has five executable statements, so SDL yields five mutants.

When applied to control structures that include a block of statements, including "*if*," "*while*," and "*for*" blocks, the entire block must be deleted. This is shown in figure 2. In the example, Mutant 1 deletes the *if* block, and Mutant 2 deletes the *for* block.

**SDL for *while* statements.** Most *while* statements include a Boolean condition and a block of statements. The condition decides whether to execute the statements in the loop. The Java SDL operator removes every statement in the loop, one at a time, then deletes the *while* condition entirely, leaving the body. To avoid compilation problems, this is accomplished by replacing the condition with *true*. The example in figure 3 has two statements in the *while* loop, so three mutants are created.

The condition is not replaced by *false*, because that would be equivalent to deleting the entire *while* statement, which is already done. The *true* mutant will often result in an infinite loop, which is handled by muJava's internal "time-out" counter.

**SDL for *if* statements.** The *if* statement is complicated by the *else* clause. The *if* statement is removed by replacing the condition with *true*, as with the *while* statement, as illustrated in figure 4, Mutant 1. Then, each statement in the body of the *if* statement is deleted (Mutants 2 and 3). Then the entire else clause is deleted (Mutant 4). The same procedure is applied to the *else* clauses, replacing the condition by true and then deleting each statement (Mutants 5, 6, and 7).

| Return Type | Mutant |
|---|---|
| int | return 0; |
| boolean | return true; / return false; |
| char | return 0; |
| double | return 0; |
| float | return 0; |
| long | return 0; |
| short | return 0; |
| String | return null; |

Table I
SDL WITH DIFFERENT RETURN TYPES

**SDL for *for* statements.** The *for* statement includes a variable declaration, a conditional statement, and an increment, each of which is treated as a statement by the SDL operator. A declaration cannot be deleted, but the conditional and the increment statements are, as shown in figure 5, Mutants 1 and 2. Then each statement inside the loop is deleted. If deleting the conditional statement results in an infinite loop, muJava's internal "time-out" counter handles it. Most of these mutants are therefore easy to kill, except when the program contains a *break* inside the loop. We did not develop rules for the *For-Each Loop* because muJava does not support Java 1.5.

**SDL for *switch* statements.** The SDL operator works at two levels for the *switch* statement. First, each *case* block is deleted, as shown in figure 6, Mutants 1, 2, and 3. Then each statement in each *case* block is deleted. Two of these are shown in figure 6, Mutants 4 and 5, and the other four are omitted from the example.

**SDL for *try-catch* blocks.** The *try-catch* blocks are complicated by Java's semantic rules for exceptions. If a statement inside a try block that could raise an exception is deleted, the corresponding catch block will result in a compiler error if the exception is checked. Thus, we also delete the corresponding catch block, unless another statement can raise the same exception inside the same try block. Figure 7 shows two examples. In Mutant 1, if the *readLine()* statement is deleted, the *IOException* catch block must also be deleted. In Mutant 2, if the *parseInt()* statement is deleted, the *NumberFormatException* catch block must also be deleted. Since SDL is a method-level mutation operator, it only considers *try-catch* blocks that are explicitly defined in the source code. Built-in exceptions are not mutated.

**SDL for *return* statements.** Deleting a *return* statement in Java would result in a compile error, so we remove the effect of the *return* statement by changing the value to the default value of the appropriate primitive type, as shown in table I.

### A. Comparison with C and Ada operators

The SDL operator defined here is more extensive than the operators that were defined for both C and Ada. The

```
public void test()
{
   int a, b;
   a = 1;
   b = 2;
}
```
Original method

```
public void test()
{
   int a, b;
   // a = 1;
   b = 2;
}
```
Mutant 1

```
public void test()
{
   int a, b;
   a = 1;
   // b = 2;
}
```
Mutant 2

Figure 1. General statement deletion mutation operator

```
public void test()
{
   int a, b, c, t;
   if (a==0)
   {
      b = 3;
   }
   for (int i = 0; i<5; i++)
      t = t + b + c;
}
```
Original method

```
public void test()
{
   int a, b, c, t;
   // if (a==0)
   // {
   //    b = 3;
   // }
   for (int i = 0; i<5; i++)
      t = t + b + c;
}
```
Mutant 1

```
public void test()
{
   int a, b, c, t;
   if (a==0)
   {
      b = 3;
   }
   // for (int i = 0; i<5; i++)
   //    t = t + b + c;
}
```
Mutant 2

Figure 2. General statement deletion mutation operator for control structures

```
public void testWhile()
{
   int a, b, c, t;
   while (a<5)
   {
      t = t + b + c;
      a++;
   }
}
```
Original method

```
public void testWhile()
{
   int a, b, c, t;
   while (true)
   {
      t = t + b + c;
      a++;
   }
}
```
Mutant 1

```
public void testWhile()
{
   int a, b, c, t;
   while (a<5)
   {
      // t = t + b + c;
      a++;
   }
}
```
Mutant 2

```
public void testWhile()
{
   int a, b, c, t;
   while (a<5)
   {
      t = t + b + c;
      // a++;
   }
}
```
Mutant 3

Figure 3. SDL for *while* statements

Ada operators [31] deleted entire control structures, but not individual statements or pieces of the control structures. This is equivalent to how we define the general SDL operator for control structures, shown in figure 2. The C operators [30] delete entire control structures, and also deletes each executable statement within the bodies of control structures. When compared with figure 3 for the *while* statement, the C operators create mutants 2 and 3. The C operators do not delete the predicate, while leaving the loop body (mutant 1 in figure 3), do not delete the individual components of the *for* loop body, do not delete individual *cases* in the *switch* statement, and do not delete the return values of *return* statements. Also, C does not have a *try-catch* block.

## IV. Empirical Evaluation of SDL-Mutation

To evaluate the effectiveness of SDL-Mutation, we implemented the specifications defined in section III in the muJava mutation system [6]. We designed tests to kill all SDL mutants for 40 subject classes, eliminating equivalent mutants by hand, then evaluated those tests by computing their mutation scores on all of muJava's method-level mutation operators.

### A. Experimental design

muJava allows testers to choose all or any subset of mutation operators to use. The SDL operator was added to the 15 existing method-level mutation operators by modifying the muJava source. This allows the SDL operator to be used in isolation, or in conjunction with other operators by making a simple selection in the GUI.

We chose 40 Java classes as experimental subjects of varying sizes, purposes, and complexity. They were taken from textbook examples and open source projects. They varied in size from 1 to 26 methods, and from 29 to 433 LOC. Statistics from all 40 are shown in table II, along with results from the study. Although fairly small, the SDL operator is intended for use during unit testing, so these are precisely the kinds of classes we would test SDL with.

Tests to kill all SDL mutants were generated by hand, by the first author. They were generated iteratively, and tests that did not (strongly) kill additional mutants were discarded. The mutants that could not be killed were analyzed for equivalence. Although not many, less than 4% of the SDL mutants were equivalent. This was a bit surprising, because it indicates the deleted statement has no affect on the program. Reasons for this are discussed in section IV-C.

Once an *SDL-adequate* set of tests was created, these tests were evaluated against **all** of muJava's mutation operators, which represent the selective set [8]. The theory is that if SDL-adequate tests can kill all mutants, then we can discard all other mutation operators and just use SDL. If the SDL-adequate tests kill very few of the other mutants, then SDL

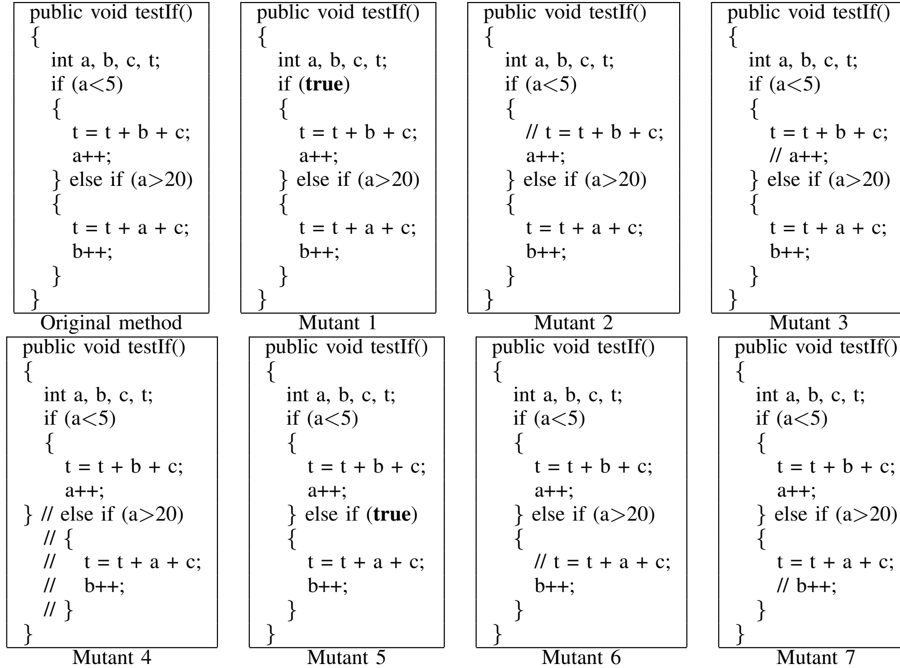


Figure 4. SDL for *if* statements

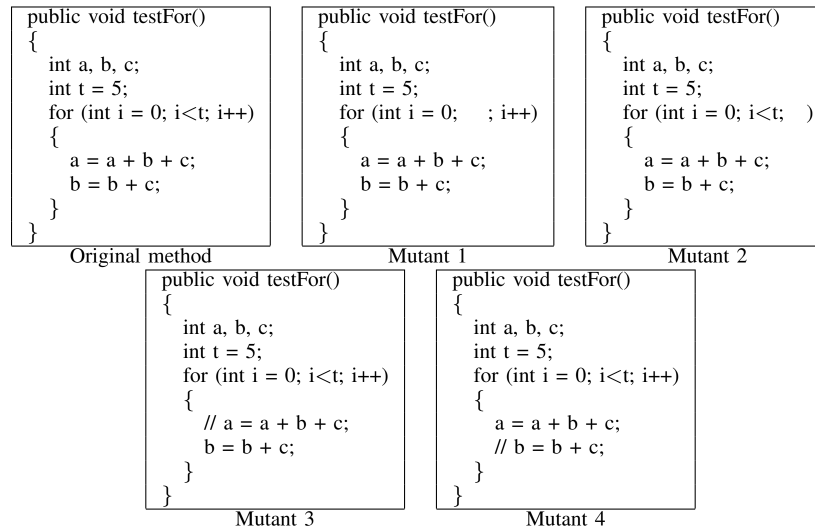


Figure 5. SDL for *for* statements

is not very useful by itself.

### B. Experimental results

Table II shows the data from the classes and the study. The first three columns give the name of the class, the number of methods, and the number of executable lines of code (*LOC*). The number of SDL mutants ranges from 4 (in *sum* and *checkIt*) 155 (in *CharArraySet*). 23 of the classes had no equivalent SDL mutants, and *BoyerMoore* had the most with 6. We designed and generated between 1 and 21 tests to kill **all** non-equivalent SDL mutants.

The four columns under **muJava** give the data from running the SDL-adequate tests on all 15 method-level muJava mutants. The number of mutants ranged from 9 (in *checkIt*) to 691 (in *CharArraySet*). Again, equivalent mutants were determined by hand by the first author from the remaining live mutants. The mutation scores of the SDL-adequate tests are shown in the last column, and ranged from a low of 0.76 (in *oddOrPos*) to 1.00 in four different classes. The mean was 0.92 and the median was 0.93.

Figure 8 gives a rough comparison of the cost. For each class, figure 8 shows the number of method-level mutants

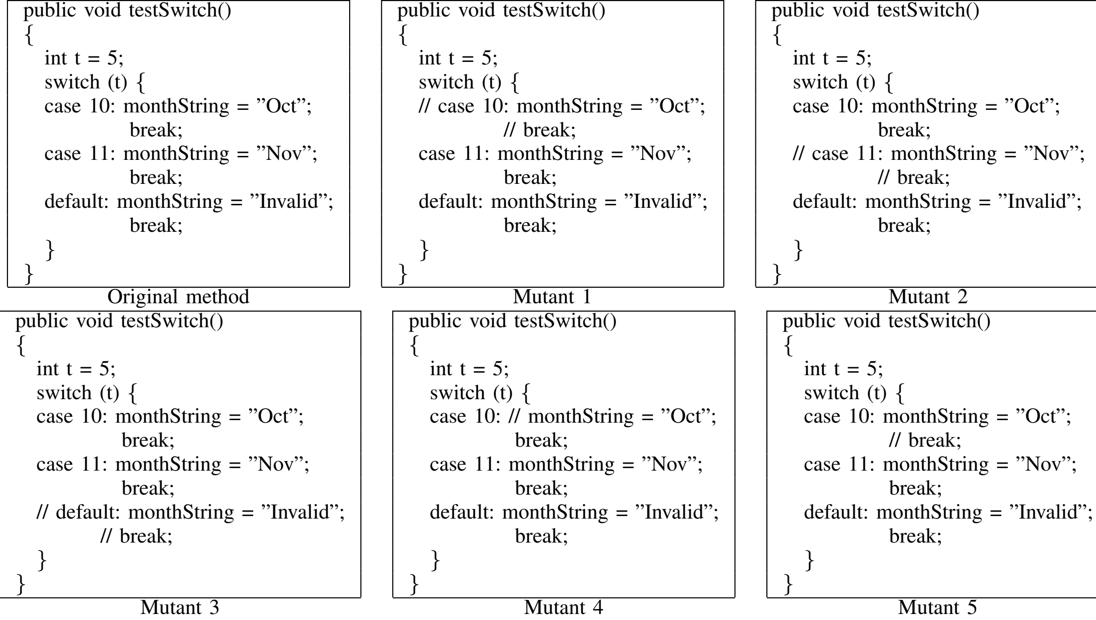


Figure 6. SDL for *switch* statements

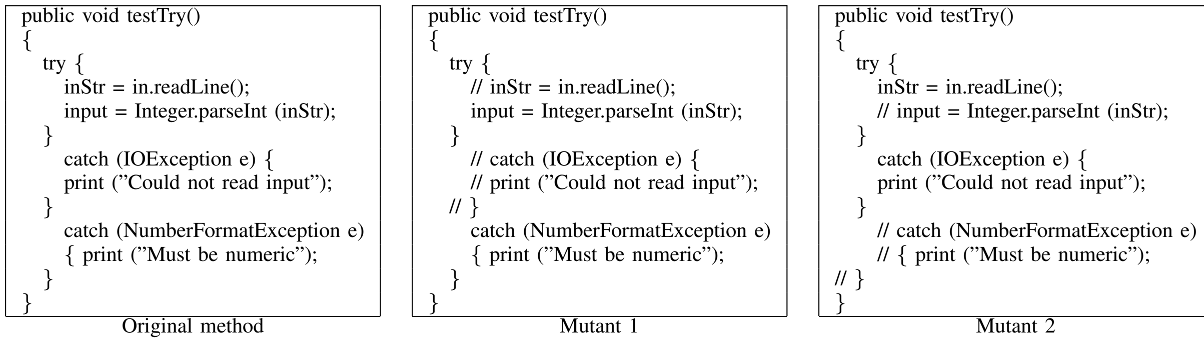


Figure 7. SDL for *try-catch* blocks

generated by muJava as the top bar (diagonal lines) and then the number of SDL mutants as the lower bar (dots).

### C. Discussion

The data in table II clearly indicate that SDL-mutation is a reasonably effective alternative to selective mutation. Although the mean mutation score of 0.92 means we cannot conclude that SDL-mutation is as strong as selective mutation, it is certainly a viable alternative, considering the savings. 90% mutation score is widely considered to be difficult to obtain [32], [33].

The direct comparison of the number of mutants for each class is shown in figure 8. The savings are impressive for all classes. Counting all classes together, we generated only 1095 SDL-mutant compared with 5807 selective mutants. This is a savings of over 81%!

Another advantage of SDL-mutation is that we have relatively fewer equivalent mutants. 9.18% of the selective mutants were equivalent, but only 3.74% of the SDL mutants. On the surface, we might expect to have zero equivalent SDL mutants, since an equivalent SDL mutant essentially means the program contains a redundant statement. Our analysis showed four reasons for redundant statements leading to equivalent SDL mutants.

First, some classes contain unnecessary control blocks. In the example in figure 9, if the *NameList* (*theNames*) is empty, the *for* loop is skipped and the empty list *NO_NAMES* is returned. However, when the *if* statement is deleted, the remaining *return* statement still returns an empty list. Thus, the *if* block is redundant. Of the 41 equivalent SDL mutants in our experiment, 36.59% (15) fall into this category.

Second, variables occasionally are given values that are never used. These assignment statements will yield equivalent SDL mutants. 24.39% of the equivalent SDL mutants had this kind of *assignment without use* statements. Third, variables are sometimes redefined with the same value before the first use. Java assigns primitive type variables

| Classes | | | SDL | | | muJava | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Methods | LOC | Mutants | Equiv | Tests | Mutants | Killed | Equiv | Score |
| booleanQuery | 23 | 433 | 31 | 0 | 10 | 94 | 74 | 12 | 0.90 |
| BoundedQueue | 6 | 82 | 33 | 4 | 9 | 184 | 151 | 18 | 0.91 |
| BoyerMoore | 5 | 107 | 47 | 6 | 3 | 328 | 283 | 13 | 0.90 |
| cal | 3 | 98 | 16 | 1 | 5 | 177 | 132 | 24 | 0.86 |
| Calculation | 9 | 116 | 32 | 1 | 9 | 271 | 182 | 45 | 0.81 |
| CharArraySet | 31 | 326 | 155 | 2 | 21 | 691 | 620 | 35 | 0.95 |
| checkIt | 2 | 29 | 4 | 0 | 2 | 9 | 7 | 0 | 0.78 |
| CheckPalindrome | 1 | 36 | 7 | 0 | 2 | 35 | 34 | 0 | 0.97 |
| Count2s | 3 | 66 | 25 | 0 | 2 | 307 | 267 | 25 | 0.95 |
| countPositive | 2 | 36 | 6 | 0 | 2 | 26 | 23 | 2 | 0.96 |
| CustomScoreProvider | 5 | 80 | 31 | 4 | 9 | 147 | 106 | 27 | 0.88 |
| Document | 16 | 211 | 68 | 2 | 15 | 170 | 143 | 11 | 0.90 |
| FieldInfos | 22 | 288 | 83 | 0 | 17 | 380 | 291 | 18 | 0.80 |
| findlast | 2 | 57 | 9 | 0 | 2 | 39 | 39 | 0 | 1.00 |
| findVal | 3 | 58 | 9 | 0 | 3 | 41 | 34 | 4 | 0.92 |
| Gaussian | 7 | 67 | 21 | 1 | 4 | 258 | 211 | 41 | 0.97 |
| GaussionElimination | 2 | 58 | 28 | 0 | 2 | 291 | 282 | 4 | 0.98 |
| Heap | 8 | 75 | 33 | 0 | 11 | 198 | 171 | 23 | 0.98 |
| IndexModifier | 26 | 344 | 22 | 0 | 15 | 63 | 50 | 8 | 0.91 |
| lastZero | 2 | 35 | 6 | 0 | 3 | 26 | 21 | 2 | 0.88 |
| LRS | 3 | 46 | 16 | 0 | 5 | 105 | 98 | 5 | 0.98 |
| MergeSort | 2 | 50 | 17 | 0 | 2 | 116 | 112 | 2 | 0.98 |
| MillisDurationField | 21 | 146 | 28 | 0 | 21 | 102 | 52 | 39 | 0.83 |
| NumberTools | 2 | 72 | 28 | 0 | 7 | 130 | 109 | 13 | 0.93 |
| numZero | 1 | 36 | 6 | 0 | 1 | 26 | 22 | 2 | 0.92 |
| oddOrPos | 2 | 36 | 6 | 0 | 2 | 46 | 32 | 4 | 0.76 |
| power | 2 | 43 | 9 | 0 | 3 | 56 | 51 | 3 | 0.96 |
| printPrimes | 3 | 54 | 20 | 1 | 3 | 73 | 63 | 6 | 0.94 |
| Queue | 6 | 78 | 29 | 4 | 8 | 150 | 125 | 12 | 0.91 |
| QuickSort | 3 | 60 | 21 | 0 | 3 | 290 | 247 | 23 | 0.93 |
| RecursiveSelectionSort | 2 | 38 | 12 | 0 | 1 | 91 | 75 | 6 | 0.88 |
| SearchString | 2 | 49 | 28 | 2 | 6 | 165 | 122 | 17 | 0.82 |
| Stack | 7 | 74 | 30 | 3 | 10 | 89 | 71 | 17 | 0.99 |
| stats | 2 | 61 | 15 | 2 | 1 | 145 | 131 | 14 | 1.00 |
| sum | 2 | 34 | 4 | 0 | 2 | 27 | 23 | 4 | 1.00 |
| Term | 13 | 123 | 44 | 3 | 16 | 57 | 49 | 8 | 1.00 |
| TermRangeQuery | 12 | 165 | 79 | 0 | 21 | 152 | 132 | 14 | 0.96 |
| TestPat | 2 | 36 | 15 | 3 | 3 | 98 | 88 | 4 | 0.94 |
| trashAndTakeOut | 2 | 37 | 15 | 1 | 4 | 117 | 88 | 24 | 0.95 |
| twoPred | 2 | 40 | 7 | 1 | 3 | 37 | 32 | 4 | 0.97 |
| **TOTAL** | **269** | **3880** | **1095** | **41** | **268** | **5807** | **4843** | **533** | |
| | | | | | | | | **Average** | **.92** |
| | | | | | | | | **Median** | **.93** |

Table II
EXPERIMENTAL RESULTS

```
private final static NameList[] NO_NAMES = new NameList[0];
public NameList[] getNameList (String goodName) {
   List result = new ArrayList();
   for (int i = 0; i <theNames.size(); i++) {
      NameList name = (NameList) theNames.get(i);
      if (name.getName().equals (goodName)) {
         result.add (name);
      }
   }
//    if (result.size() == 0)
//       return NO_NAMES;
   return (NameList[]) result.toArray (new NameList [result.size()]);
}
```

Figure 9. Example of SDL equivalent mutant

a default value at their declarations. Programmers will often follow that with an assignment of the same value, resulting in a redundant statement and an equivalent SDL mutant. 21.95% of the equivalent SDL mutants were *redefine* statements. Fourth, *break* statements sometimes yield equivalent mutants. Programmers often insert unnecessary *break*s into *case* statements as an engineering measure to add redundancy, and to protect against future changes. 9.76% of the equivalent SDL mutants were *break* statements.

Simple slicing techniques [34] could probably find all or most of these equivalent mutants. In fact, optimizing compilers often eliminate these kinds of redundant statements, so the same technique could be used to eliminate equivalent SDL mutants [35].

SDL did relatively poorly on some classes, including six where the SDL-adequate tests had less than an 85% mutation score and two with less than an 80% score. A hand analysis of the remaining live mutants showed three general reasons

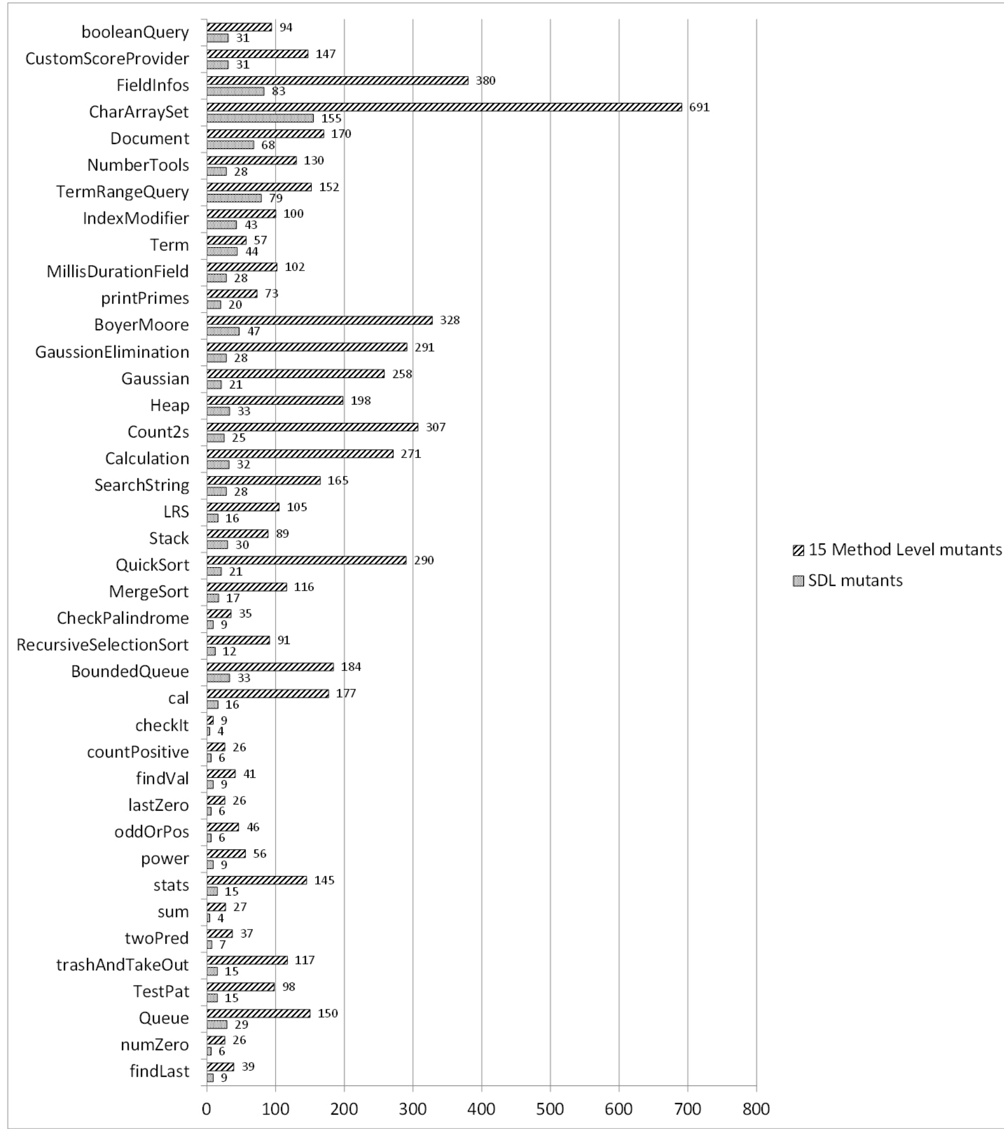


Figure 8. Comparison of SDL mutants with muJava's method level mutants

for SDL-mutation being weaker. First is the size; if the class does not have many mutants, then one or two mutants that cannot be killed will have a big impact on the ratio. The class *checkIt* falls into this category. The second reason is a class having lots of arithmetic or logical operations. SDL mutants do not result in tests that explore the input space, particularly boundary values, that will kill as many of those mutants. The third is related; if the class has statements with complicated operations, such as bit shifting, SDL mutants are not likely to yield values that are subtle enough to kill 90% of these mutants. The classes in table II with particularly low mutation scores exhibited one or more of these characteristics.

### D. *Threats to validity*

This experiment has several threats to validity, which we tried to ameliorate as much as possible. First the results are all on Java programs. The implementation of the SDL operator would be different in different languages, which could affect the results. This could only be managed by implementing the SDL operator on other languages, a task we leave for the future. As with any software engineering experiment, there is no way to know whether the subjects are representative. We tried to choose classes from a variety of sources, sizes, and applications.

As mentioned before, equivalent mutants were determined by hand by one person. This could have introduced errors into the process, although the affect on the results would be small. This experiment used only one test set per subject,

which could result in some unusually low or high scores. We believe a low score is just as likely as a high score, so with 40 subjects, this threat should be balanced. Moreover, experimenters who have previously used multiple test sets per subject have reported very little deviation in the results among the test sets [36].

Finally, it is possible that the implementation of the SDL operator in muJava could be faulty. To reduce this threat, we checked results quite carefully by hand; examining SDL mutants for all types of statements.

## V. Conclusions and Recommendations

This research started with the theory, first proposed by Untch [12], that performing mutation testing with only one very selective operator, statement deletion (SDL), will result in very effective tests. This theory is attractive because *SDL-mutation* has the potential to be significantly cheaper than mutation with "all" operators (*standard mutation*), or mutation with the selective set of operators (*selective mutation*).

We evaluated this theory by first defining SDL for Java, a task that is somewhat complicated because the many advanced control structures forced us to think hard about the meaning of both "delete" and "statement." We then implemented the resulting SDL operator in the muJava mutation system to facilitate a direct comparison with muJava's selective mutation approach.

In the experiment, we killed all SDL mutants for 40 classes, resulting in a suite of SDL-adequate tests. Then we ran the SDL-adequate tests on all of muJava's mutants. The results were that SDL-adequate tests scored an average mutation score of **92**, with **80%** fewer mutants. We also found **41%** fewer equivalent SDL mutants. This is a huge savings with only a modest loss in effectiveness.

We recommend that future mutation systems include SDL-mutation as a choice, not a replacement. The new version of muJava (version 4) includes SDL[1]. We recommend a mutation system that has a "strength dial," which can be turned to "low" to choose SDL-mutation or "strong" to choose selective mutation. This could be mixed with weak mutation or other approaches that decrease the cost of mutation.

Research into SDL-mutation is not finished. Even though SDL is intended to be used during unit testing, more experiments with larger subjects should be carried out. Also, as mentioned in Section III, muJava and SDL as defined in this paper are specific to Java. SDL should also be applied to other languages so that we can compare results across different languages. This study should also be replicated with other programs. Another replication would be to compare the ability to kill SDL mutants with the same number of randomly selected mutants. Furthermore, we would like to see a comprehensive fault study to determine how many actual faults SDL-mutation detects when compared with selective mutation.

We also believe a further refinement may result in further savings. It seems likely that we could use slicing techniques to show that certain tests that kill the SDL mutant for one statement are **guaranteed** to kill the SDL mutant for some subsequent statements. Finally, we do not know how SDL-mutation interacts with approaches such as weak mutation. This would need to be determined empirically.

[1] http://www.cs.gmu.edu/∼offutt/mujava/

## VI. Acknowledgment

We would like to thank Roland Untch of Middle Tennessee State University for suggesting the idea of SDL-mutation.

## 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions of Software Engineering*, vol. 37, no. 5, pp. 649–678, September 2011.

[8] 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.

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

[10] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis," School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, Technical Report GIT-ICS-79/08, September 1979.

[11] 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.

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

[13] R. A. DeMillo, H. Pan, and G. Spafford, "Critical slicing for software fault localization," in *Proceedings of the 1996 International Symposium on Software Testing, and Analysis*. San Diego, CA: ACM Press, January 1996, pp. 121–134.

[14] R. Untch, "Schema-based mutation analysis: A new test data adequacy assessment method," Ph.D. dissertation, Clemson University, Clemson SC, 1995, clemson Department of Computer Science Technical report 95-115.

[15] 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*, San Jose, CA, October 2000, pp. 45–55.

[16] W. E. Wong, "On mutation and data flow," Ph.D. dissertation, Purdue University, December 1993, (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

[17] 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.

[18] 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.

[19] 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*. New York, NY, USA: ACM, 2008, pp. 351–360.

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

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

[23] 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*. Banff, Alberta: IEEE Computer Society Press, July 1988, pp. 171–177.

[24] 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*, Chicago, Illinois, August 1992, pp. II–257–266.

[25] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 371–379, July 1982.

[26] J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, May 1994.

[27] B.-J. Choi and A. P. Mathur, "High-performance mutation testing," *Journal of Systems and Software, Elsevier*, vol. 20, no. 2, pp. 135–152, February 1993.

[28] 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)*, Kogakuin University. Tokyo, Japan: IEEE Computer Society Press, September 1991.

[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*, Cambridge MA, June 1993, pp. 139–148.

[30] 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.

[31] J. Offutt, J. Payne, and J. M. Voas, "Mutation operators for Ada," Department of Information and Software Engineering, George Mason University, Fairfax VA, Technical Report ISSE-TR-96-09, March 1996, http://www.cs.gmu.edu/∼tr_admin/.

[32] 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*. Montreal, Canada: IEEE Computer Society Press, June 1991, pp. 410–417.

[33] J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software-Practice and Experience*, vol. 26, no. 2, pp. 165–176, February 1996.

[34] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification, and Reliability, Wiley*, vol. 9, no. 4, pp. 233–262, December 1999.

[35] J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification, and Reliability, Wiley*, vol. 4, no. 3, pp. 131–154, September 1994.

[36] P. G. Frankl and Y. Deng, "Comparison of delivered reliability of branch, data flow and operational testing: A case study," in *Proceedings of the 2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*. Portland OR: IEEE Computer Society Press, August 2000, pp. 124–134.