e-Informatica Software Engineering Journal, Volume 10, Issue 1, 2016, pages: 31–49, DOI 10.5277/e-Inf160102

Reducing the Number of Higher-order Mutants with the Aid of Data Flow

Ahmed S. Ghiduk[a]

[a] *Dept. of IT, College of Computers and Information Technology, Taif University, Saudi Arabia, Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Egypt*

asaghiduk@tu.edu.sa

Abstract

Higher-order mutants are created by injecting two or more mutations into the original program, while first-order mutants are generated by seeding single faults in the original program. Mutant generation is a key stage of mutation testing which is computationally very expensive, especially in the case of higher-order mutants. Although many mutation testing techniques have been developed to construct the first-order mutants, a very small number of techniques have been presented to generate the higher-order mutants because of the exponential growth of the number of higher-order mutants, and the coupling effect between higher-order and first-order mutants. To overcome the exponential explosion in the number of higher-order mutants considered, this paper introduces a new technique for generating a reduced set of higher-order mutants. The proposed technique utilizes a data-flow analysis to decrease the number of mutation points through the program under test and consequently reduce the number of higher-order mutants. In this technique only positions of *defs* and *uses* are considered as locations to seed the mutation. The generated set of higher-order mutants consists of a reduced number of mutants, which reduces the costs of higher-order mutation testing. In addition, the proposed technique can generate the higher-order mutants directly without generating the first-order mutants or by combining two or more first-order mutants. A set of experiments are conducted to evaluate the effectiveness of the proposed technique. The results of the conducted experiments are presented and compared with the results of the related work. These results showed that the proposed technique is more effective than the earlier techniques in generating higher-order mutants without affecting the efficiency of mutation testing.

Keywords: mutation testing, first-order mutants, higher-order mutants, data-flow analysis

1. Introduction

Higher-order mutants (*HOMs*) are complex mutants, which are produced by inserting two or more mutations in the original program [1]. The space of higher-order mutants is wider than the space of first-order mutants (*FOMs*) [2]. Mutation testing has been developed by DeMillo et al. [3] and Hamlet [4] to find test inputs to kill the seeded mutants in the program under test [5]. The motivation of mutation testing is that the injected faults represent errors that programmers often create. Although mutation testing is a very powerful software testing technique, it contains many computationally expensive phases such as mutant generation and mutant execution.

Many mutation testing techniques have been developed to consider the first-order mutants [6]. Higher-order mutation testing techniques are proposed by Jia and Harman [1] and used to study the interactions between defects and their impact on software testing for fault detection.

Although mutation testing is an effective high automation technique to assess the quality of the test data, it has three main limitations. These limitations are large number of mutants, realism, and the equivalent mutant problem [7,8]. A large number of mutants will be generated during the

mutant generation phase of mutation testing even for small programs. For example, a program consists of one statement such as *return* $x + y$; (where x and y are integers) can be mutated into many different mutants: *return* $x-y$; *return* $x*y$; *return* x/y; *return* $x + y + +$; *return* $-x + y$; *return* $x + -y$; *return* $0 + y$; *return* $x + 0$; ..., etc. This problem leads to a very high execution cost because the test cases are executed not only on original program but also on each mutant. For example, if a program under test has 200 mutants and 150 test cases, it requires $(1+200)*150 = 30150$ executions with their corresponding results [7]. In addition, because mutants are generated by single and simple syntactic changes, they don't represent realistic faults and 90% of real faults are complex [2]. In fact, several mutation operators can generate equivalent mutants which have the same behavior as the original program and need additional human effort to kill [9]. These limitations are resulting from the used method to generate mutants.

Many techniques have been proposed to reduce the number of mutants. The first approach to reduce the number of mutants is Mutant Sampling approach proposed by Acree [10] and Budd [11]. In addition, Bluemke and Kulesza [12] explored the reduction of computational costs of mutation testing by randomly sampling mutants. This approach randomly selects a small subset of mutants from the entire set. Mutant Sampling is valid with a value higher than 10% of mutants [7]. Agrawal et al. [13] and Mathur [14] proposed an approach to reduce the number of mutation operators which can lead to reduced number of mutants. Offutt et al. [15, 16] used the same idea and called it Selective Mutation. This approach selects a small set of operators that generate a subset of all possible mutants without losing test effectiveness. Husain [17] applied clustering algorithms to select a subset of mutants.

Second-order Mutation Testing [9, 18–20], in particular, and Higher-Order Mutation Testing [1, 2, 21, 22] in general, are the most promising solutions to reduce the number of mutants [7]. The number of generated mutants can be reduced to about 50% by combining two first-order mutants to generate a second-order mutant or by using subsuming higher-order mutants algorithms [7]. Previous work employed different methods for reducing the number of higher-order mutants. Polo et al. [20] proposed three methods: 1) *RandomMix* which couples randomly selected mutation operators, 2) *LastToFirst* which combines *FOMs* in order from the last operator to the first one, and 3) *DifferentOperator* which combines different *FOMs* mutation operators. Madeyski et al. [9] proposed two methods: 1) *JudyDiffOp* which combines different *FOMs* mutation operators, and 2) *NeighPair* which combines *FOMs* which are close to each other. Although, these techniques have the ability to reduce the number of mutants, the number of mutants can still grow exponentially. From the above discussion, the higher-order mutant generation problem needs a lot of effort.

Data-flow testing is essential because it augments control-flow testing. It aims at creating more efficient and targeted test suites. Data flow testing is concerned not only with the definitions and uses of variables, but also with sub-paths from definitions to statements where those definitions are used [23, 24]. A family of data flow criteria [25] have been proposed and successfully applied in many software testing activities [26]. Unfortunately, this family of criteria has never been applied in mutation testing as basis for generating the *HOMs* or reducing the number of these mutants.

The main contributions of this paper are: 1) introducing a data-flow based approach for generating higher-order mutants; In this approach only locations of *def* points and *use* points are considered as locations to seed the mutation. A second-order mutant contains two mutations, the first mutation at the *def* point and the second mutation at the *use* point and the two points belong to the same *def-use* pairs. 2) Using the proposed approach to perform a set of empirical studies to answer the following research questions:

- **RQ1**: How effective is data flow in aiding the generation of higher-order mutants?
- **RQ2**: How effective is the proposed technique in reducing the number of higher-order mutants?

Table 1. An example of mutants

Original Program	Mutants		
	FOM1	*FOM2*	*SOM*

if (min < max)
{
max = min + max;
min = max − min;
}
	if (min > max)
	{
	max = min + max;
	min = max − min;
	}
		if (min < max)
		{
		max = min − max;
		min = max − min;
		}
			if (min > max)
			{
			max = min − max;
			min = max − min;
			}

The rest of this paper is organized as follows. Section 2 gives some basic concepts and definitions. Section 3 describes the proposed technique for generating a reduced set of higher-order mutants. Section 4 describes the empirical studies performed to evaluate the proposed technique. Section 5 gives a discussion of how the present paper differs from the related ones. Section 6 gives the conclusion and future work.

2. Background

This section introduces some basic concepts that will be used throughout this work.

2.1. Mutation Testing

The input parameters to the mutation testing are: the tested program P, a set of mutation operators, and a set of test inputs, T. Initially, the program under test must be executed with the test set T to show that it is correct and produces the desired outputs. If not, then the program under test contains faults, which should be corrected before resuming the process.

The next stage is generating a set of mutants of the tested program by seeding faults in it. The seeded faults are generated by applying the mutation operators. The transformation that creates a mutant from the original program is known as a mutation operator [1]. A mutant is generated by making one or more small changes (faults) into the original program. *FOMs*, which are created by the injection of unique faults in the tested program, are created by applying mutation operators only once. *HOMs*, which are created by injecting two or more mutations into the original program, are created by applying mutation operators more than once. Table 1 shows two first-order mutants (*FOM1* and *FOM2*) generated by changing the "<" operator in the original program into the ">" operator in *FOM1* and changing the "+" operator in the original program into the "−" operator in *FOM2*. In addition, Table 1 gives a second-order mutant, *SOM*, created by coupling the two first-order mutants *FOM1* and *FOM2*.

In Traditional Mutation Testing (Strong Mutation), each mutant will be executed using a test set T. If the result of executing a mutant is different from the result of executing the original program for any test case in T, then the mutant is *killed* otherwise it is *survived*. The adequacy level of the test set T can be measured by a mutation score [27] that is computed in terms of the number of mutants killed by T as follows.

$$\mathrm{MS}(P,T) = \frac{\text{Number of Killed Mutants}}{\text{Total Number of Mutants} - \text{Number of Equivalent Mutants}} \tag{1}$$

Howden [6] proposed Weak Mutation [28] to optimize the execution of Strong Mutation. Weak Mutation checks the result of a mutant immediately after the mutated component is executed with the resulting execution of the original component to say is the mutant is killed or not.

2.2. Higher-order Mutation Testing

Higher-order mutation (*HOM*) testing is a generalization of traditional mutation testing. Higher-order mutants are constructed by inserting two or more changes into the program under test or by combining two or more first-order mutants. Higher-order mutants can be classified into six categories based on the way that they are *Coupled* and *Subsuming* [1]. Coupled means: complex errors are coupled to simple errors, and the coupling effect hypothesis states that test input sets that detect simple types of faults are sensitive enough to detect more complex types of faults [3]. A subsuming *HOM* is one in which the first-order constituent mutants partly mask one another. Therefore, a subsuming *HOM* is harder to kill than the first-order mutants from which it is constructed.

2.3. Data-flow Analysis

The structure of the program can be represented by the control-flow graph. A control-flow graph $G = (N, E)$ with a unique entry node n_0 and a unique exit node n_k, consists of a set N of nodes, where each node represents a statement, and a set E of directed edges, where a directed edge $e = (n, m)$ is an ordered pair of two adjacent nodes, called *tail* and *head* of e, respectively [29,30].

Data-flow analysis identifies all definition-use (*def-use*) pairs for any variable v of the program under test. A *def-use* is the order triple (d, u, v) in which statement d contains a *definition* for variable v and statement u contains a *use* of v that can be reached by d over some paths in the program under test [23, 24]. A variable is defined in a statement when its value is assigned or changed. A variable is used in a statement when its value is utilized in a statement and not changed. A *predicate use* (*p-use*) for a variable indicates the *use* of the variable in a predicate. A *computational use* (*c-use*) indicates the *use* of the variable in a computation.

3. A Proposed Higher-Order Mutant Generation Technique

This section describes the proposed technique for generating a reduced set of higher-order mutants. This technique utilizes the concepts of data-flow analysis of the program to reduce the number of mutation positions through the tested program which will reduce the number of higher-order mutants. The proposed technique is based on data-flow analysis [31] and Muclipse tool [32,33]. The proposed technique consists of the following main modules.

1. Analysis Module.
2. Mutant Generation Module.
3. Mutant Filtering Module.

These modules are described in more detail below.

3.1. Analysis Module

This module applies the data-flow analysis procedure proposed by F.E. Allen and J. Cocke [31] to find all definition-use pairs (all definition-c-use and all definition-p-use) in the tested program. This module reads the Java source code of the program under test, builds the control-flow graph of the tested program, and identifies all definition-use pairs for each method in this Java program individually. The proposed technique reduces the number of *def-p-use* pairs by combining all *def-p-use* pairs which have the same *def* point (i.e., beginning statement of the edge *p-u*) into one *def-p-use* pair where the *use* point (end statement *u*) does not contain any *uses*. The outputs of this phase are passed to the Mutant Generation Module (step 2).

For the Java example program shown in Table 2, this phase finds all definition-c-uses and all definition-p-uses pairs in the tested program by applying the proposed data-flow analysis procedure. The Analysis Module finds 10 *def-c-use* pairs and 20 *def-p-use* pairs for method Midnum(). Table 3 shows all *def-use* pairs of method Midnum() of the example program given in Table 2. In Table 3, a *def-c-use* (*d, cu, x*) consists of the statement *"d"* which contains a definition for variable *"x"* which is used in a computation state-

Table 2. Java example program

1. package edu.ncsu.csc326.paperHOM_dataflow;	29. {
2. public class Mid1 {	30. mid = y;
3. private int num1, num2, num3, Mid;	31. }
4. public Mid1(){	32. else
5. }	33. {
6. public void setNum1(int x){	34. if(x<z)
7. num1 = x;	35. {
8. }	36. mid = x;
9. public void setNum2(int x){	37. }
10. num2 = x;	38. }
11. }	39. }
12. public void setNum3(int x){	40. else
13. num3 = x;	41. {
14. }	42. if(x>=y)
15. public int getMid(){	43. {
16. return Mid;	44. mid = y;
17. }	45. }
18. public void Midnum()	46. else
19. {	47. {
20. int x, y, z;	48. if(x>z)
21. int mid;	49. {
22. x = num1;	50. mid = x;
23. y = num2;	51. }
24. z = num3;	52. }
25. mid = z;	53. }
26. if(y<z)	54. Mid = mid;
27. {	55. }
28. if(x<y)	56. }

ment *"cu"* and a *def-p-use* (*d, p-u, x*) consists of the statement *"d"* which contains a definition for variable *"x"* which is used through the edge *"p-u"* which starts at statement *"p"* and ends at statement *"u"*. Then, the proposed technique reduces the number of *def-p-use* pairs by merging all *def-p-use* pairs which have the same *def* point. Therefore, two *def-p-use* pairs such as (*22, 28-39, x*) and (*22, 28-32, x*) will merge to one *def-p-use* (*22, 28, x*). The proposed technique reduces the 20 *def-p-uses* to 10 *def-p-uses*. Table 3 gives the new *def-p-uses* pairs. The list of *def-c-uses* and the reduced *def-p-uses* are passed to the Mutant Generation Module.

3.2. Mutant Generation Module

This module uses the data collected by the Analysis Module to generate the set of higher-order mutants. This module considers only the locations of *def* points and *use* points as locations to seed mutation. This module uses the set of method-level operators proposed by Y. Ma and J. Offutt [34] using the Muclipse tool [32, 33] to generate the first-order mutants. Table 4 shows this set of mutation operators. This module requires three inputs to perform its task (i.e., generating a set of higher-order mutants): the first input is the Java source code of the program under test, the second is the set of mutation operators given in Table 4, and the third is the set of mutation locations which is the location of *def* and *use* statements (i.e., set of *defs* ∪ set of *uses*) in the tested program. For the example program given in Table 2, the set of mutation locations is $\{22, 23, 24, 25, 30, 36, 44, 50\} \cup \{36, 50, 30, 44, 25, 54, 28, 34, 42, 48, 26\} = \{22, 23, 24, 25, 26, 28, 30, 34, 36, 42, 44, 48, 50, 54\}$.

For generating first-order mutants, the proposed technique needs a set of mutation operators, a set of mutation locations, and the program to be mutated. For the example pro-

Table 3. All def-uses of method Midnum() of the example program

#	def-c-uses	def-p-uses		Reduced def-p-uses
1	(22,36,x)	(22,28-39,x)	(22,28-32,x)	(22,28,x)
2	(22,50,x)	(22,34-35,x)	(22,34-38,x)	(22,34,x)
3	(23,30,y)	(22,42-43,x)	(22,42-46,x)	(22,42,x)
4	(23,44,y)	(22,48-49,x)	(22,48-52,x)	(22,48,x)
5	(24,25,z)	(23,26-27,y)	(23,26-40,y)	(23,26,y)
6	(25,54,mid)	(23,28-29,y)	(23,28-32,y)	(23,28,y)
7	(30,54,mid)	(23,42-43,y)	(23,42-46,y)	(23,42,y)
8	(36,54,mid)	(24,26-27,z)	(24,26-40,z)	(24,26,z)
9	(44,54,mid)	(24,34-35,z)	(24,34-38,z)	(24,34,z)
10	(50,54,mid)	(24,48-49,z)	(24,48-52,z)	(24,48,z)

Table 4. Set of mutation operators

Category	Mutation Operator	Description
AO	AORB	A binary arithmetic operator is replaced by another one.
	AORU	An unary arithmetic operator is replaced by another one.
	AORS	A short-cut arithmetic operator is replaced by another one.
	AOIS	A short-cut arithmetic operator is inserted into the program.
	AOIU	An unary arithmetic operator is inserted into the program.
	AODS	A short-cut arithmetic operator is deleted from the program.
	AODU	An unary arithmetic operator is deleted from the program.
RO	ROR	A relational operator is replaced by another one.
CO	COR	A binary conditional operator is replaced by another one.
	COI	An unary conditional operator is inserted into the program.
	COD	An unary conditional operator is deleted from the program.
SO	SOR	A shift operator is replaced by another one.
LO	LOR	A binary logical operator is replaced by another one.
	LOI	An unary logical operator is inserted into the program.
	LOD	An unary logical operator is deleted from the program.
AS	ASRS	A short-cut assignment operator is replaced by another one.

gram, the set of mutation locations is the set of *defs* locations and *uses* locations $= \{22, 23, 24, 25, 26, 28, 30, 34, 36, 42, 44, 48, 50, 54\}$ and the set of mutation operators is the 16 operators given in Table 4. The following pseudocode presents the proposed *DataFolwBasedFOM* algorithm for generating a reduced list of *FOMs.*

```
Algorithm DataFolwBasedFOM(program,
          mutationPoints[], operators[])
LET firstOrderMutants be an empty list
WHILE mutationPoints.size() > 0 DO
  WHILE !(oprators.empty()) DO
    op = oprators.select();
    mp = mutationPoints.select();
    newMutant = program.mutate(op, mp);
    firstOrderMutants.update(newMutant);
  ENDWHILE
ENDWHILE
RETURN firstOrderMutants;
```

The function *Operator.select*() uses different procedures to select an operator such as 1) not selected yet, 2) different operator, and 3) different category and the function *mutationPoints.select*() selects a not selected yet mutation position.

A second-order mutant contains two mutations, the first mutation at the *def* position and the second mutation at the *use* position and

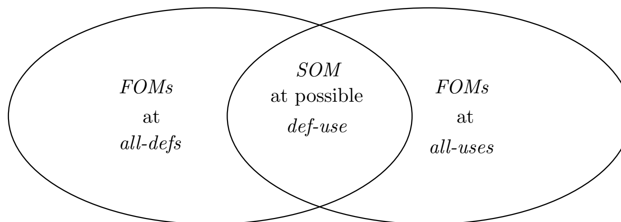

Figure 1. Mathematical representation of *FOM* and *SOM*

the two positions belong to the same *def-use* pairs. Therefore, the set of second-order mutants is the intersection between the set of first-order mutants at *defs* locations with the set of first-order mutants at *uses* locations. To generate second-order mutants, the proposed technique inserts two mutations into the original program at the *def* and the *use* locations of the same *def-use* pairs. It can also merge the two first-order mutants at *def* and *use* positions of the same *def-use* pairs to construct a second-order mutant. Figure 1 shows a mathematical representation for the first-order mutants and the second-order mutants.

According to the above description the proposed technique needs only mutation positions, mutation operators, and the program to be mutated to generate higher-order mutants without needing the first-order mutants. The proposed technique selects one of the elements of the set of all *def-use* pairs and seeds this element with two mutation operators: one operator is applied at the *def* location and the second operator is applied at the *use* location. The following pseudocode presents the proposed *DataFolwBasedSOM* algorithm for generating the reduced list of second-order mutants.

```
Algorithm DataFolwBasedSOM(program,
         allDefUsePairs[], operators[])
LET secondOrderMutants be an empty list
WHILE allDefUsePairs.size()>0 DO
  WHILE !(oprators.empty()) DO
    op1 = oprators.select();
    op2 = oprators.select();
    du= allDefUsePairs.select();
    newMutant=program.mutate(op1, op2,
                             du);
    secondOrderMutants.update(newMutant);
  ENDWHILE
ENDWHILE
RETURN secondOrderMutants;
```

The function *Operator.select*() uses different procedures to select two operators such as 1) not selected yet, 2) different operator, and 3) different category and the function *allDefUsePairs.select*() selects a not selected yet *def-use* pairs to be a mutation point.

Table 5 (a) presents an example for second-order mutant of the example program at the *def-c-use* (*22, 36, x*) and Table 5 (b) presents an example for second-order mutant of the example program at the *def-p-use* (*24, 48, z*).

For generating higher-order mutants of even order greater than the second-order, the proposed technique applies the *DataFolwBasedSOM* algorithm more than one time with a change of the input program to the output or mutated program of the previous cycle. To generate higher-order mutants of odd order greater than the second order, the proposed technique applies the *DataFolwBasedSOM* algorithm more than one time with a change of the input program to the output or mutated program of the previous cycle in such a way that in the last cycle the algorithm seeds one mutation operator at *def* or *use* location only.

For example, for generating fourth-order mutants the technique applies the *DataFolwBasedSOM* algorithm two times in such a way that the inputs of the second cycle are the mutated programs (second-order mutants) of the first cycle. To generate third-order mutants, the technique applies the *DataFolwBasedSOM* algorithm two times such that the inputs of the second cycle are the mutated programs of the first cycle (second-order mutants) restricting the function *program.mutate*(*op1, op2, du*) to seed one operator at the *def* location or at the *use* location only.

Table 5. An example for second-order mutant of the example program

18. public void Midnum()	38. }	18. public void Midnum()	38. }
19. {	39. }	19. {	39. }
20. int x, y, z;	40. else	20. int x, y, z;	40. else
21. int mid;	41. {	21. int mid;	41. {
22. **x = ++num1;**	42. if(x>=y)	22. x = num1;	42. if(x>=y)
23. y = num2;	43. {	23. y = num2;	43. {
24. z = num3;	44. mid = y;	24. **z = num3++;**	44. mid = y;
25. mid = z;	45. }	25. mid = z;	45. }
26. if(y<z)	46. else	26. if(y<z)	46. else
27. {	47. {	27. {	47. {
28. if(x<y)	48. if(x>z)	28. if(x<y)	48. **if(x<z)**
29. {	49. {	29. {	49. {
30. mid = y;	50. mid = x;	30. mid = y;	50. mid = x;
31. }	51. }	31. }	51. }
32. else	52. }	32. else	52. }
33. {	53. }	33. {	53. }
34. if(x<z)	54. Mid = mid;	34. if(x<z)	54. Mid = mid;
35. {	55. }	35. {	55. }
36. **mid *= x;**	56. }	36. mid = x;	56. }
37. }		37. }	
(a) SOM at du-pair (22,36,x)		(b) SOM at du-pair (24,48,z)	

Table 6 gives examples for third-order (*3OMs*) and fourth-order (*4OMs*) mutants of the example program given in Table 2.

3.3. Mutant Filtering Module

This module eliminates any useless mutants from the generated set of higher-order mutants. This module uses some criteria to divide the mutants into two categories: the first category is the target set of *HOMs* and the second one is the set of useless mutants. These criteria are:

1. Redundant mutants: the repeated mutants which were generated before.
2. First-order mutants: this happens if the mutation location of the *SOM* refers to the same position of the *FOM* (i.e., the same arithmetic operator in the same statement). This happens if the *def* location and *use* location are in the same statement. For example in the following loop:

```
1 i = 0;
2 sum = 0;
3 while(i < 10)
4    sum = sum + i;
```

   In the above code, the *def-use* (*4, 4, sum*) is a *def-use* at statement 4 for variable *sum.* In this *def-use* pairs, the *def* location and the *use* location are the same. Therefore, the proposed algorithm can generate a first-order mutant by changing the addition operator " + " to the division operator " / " and changing the division operator " / " to the addition operator " + ".
3. Equivalent mutants: this module can be supported by a technique for identifying the equivalent mutants to remove it. In our experiments, equivalent mutants are manually identified.

4. Empirical Studies

This section describes the empirical studies performed to evaluate the proposed technique. Two empirical studies were conducted: the first study aims to investigate the efficiency of data flow in aiding the generation of higher-order mutants and reducing their number as well; the second study aims to demonstrate that the proposed mutants do not lead to a substantial loss in the effectiveness of the method.

Table 6. An example for third and fourth order mutants of the example program

18. public void Midnum()	38. }	18. public void Midnum()	38. }
19. {	39. }	19. {	39. }
20. int x, y, z;	40. else	20. int x, y, z;	40. else
21. int mid;	41. {	21. int mid;	41. {
22. **x = ++num1;**	42. if(x>=y)	22. x = num1;	42. if(x>=y)
23. y = num2;	43. {	23. y = num2;	43. {
24. z = num3;	44. mid = y;	24. **z = num3++;**	44. mid = y;
25. mid = z;	45. }	25. mid = z;	45. }
26. if(y<z)	46. else	26. if(y<z)	46. else
27. {	47. {	27. {	47. {
28. if(x<y)	48. if(x>z)	28. if(x<y)	48. **if(x<z)**
29. {	49. {	29. {	49. {
30. **mid = y++;**	50. mid = x;	30. **mid = ++y;**	50. mid = x;
31. }	51. }	31. }	51. }
32. else	52. }	32. else	52. }
33. {	53. }	33. {	53. }
34. if(x<z)	54. Mid = mid;	34. if(x<z)	54. **Mid = -mid;**
35. {	55. }	35. {	55. }
36. **mid *= x;**	56. }	36. mid = x;	56. }
37. }		37. }	
(a) 3OM at du-pairs (22,36,x), and (30,54,mid)		(b) 4OM at du-pair (24,48,z) and (30,54,mid)	

4.1. Empirical Study #1

4.1.1. Setup of Empirical Study #1

Prototype: Figure 2 gives the architecture of the prototype *HOMG*, which consists of three modules: an analysis module, a mutant generation module, and a mutants filtering module. This prototype is based on the proposed technique which is presented in Section 3.

Subject Programs: A set of Java programs was selected from the previous studies for conducting an empirical study to evaluate the proposed technique. The set of subject programs contains some common programs which are often used as benchmarks in many software testing studies. This set of programs is *triangle*, *mid*, *power*, *remainder*, and three synthetic programs with different and complex structures.

Table 7 presents the details of the subject programs: the first column, *Subject Program*, presents a designated title of the program under test; the second column, *Reference*, presents some of the previous studies which used this set of subject programs; and the third column, *Scale*, presents the number of lines of code, classes, and methods in the subject program.

Procedure: the empirical study is conducted as follows.

1. Run *Muclipse* tool on the program to be mutated (original program) to generate *FOMs*. Because *Muclipse* cannot generate second-order mutants mutants the Muclipse tool was run on each first-order mutant to generate all possible second-order mutants. This set of second-order mutants is used for comparing the *LastToFirst* Algorithm, *DifferentOperators* Algorithm, and our proposed Algorithm.
2. Run the analysis module of our technique to find all *def-use* pairs of the original program.
3. Run the mutants generation module according to the *DataFolwBasedSOM* algorithm. Then the useless mutants are removed.

4.1.2. Objectives of Study #1

The study procedure to measure the efficiency of our proposed technique in generating the second-order mutants was applied. This study addresses the following research questions:

- **RQ1**: How effective is data flow in aiding the generation of higher-order mutants?

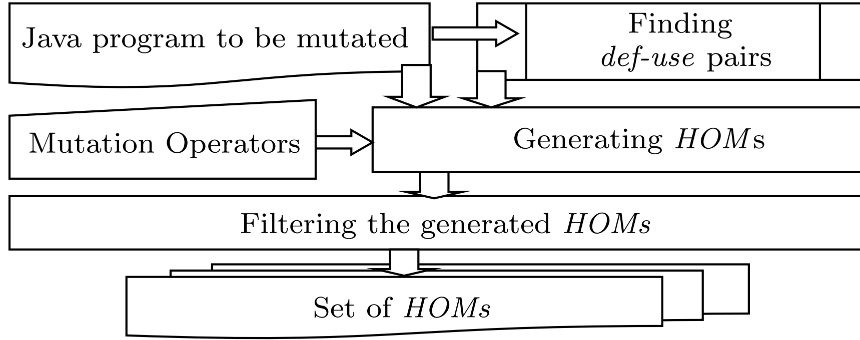

Figure 2. The architecture of the prototype of *HOMG*

Table 7. Subject programs

# Subject Program	Reference	Scale
P#1. Triangle	[2,20,26,35]	73 LOC, 1 C, 6 M
P#2. Mid	[20,26,36]	61 LOC, 1 C, 6 M
P#3. Power	[26,36,37]	49 LOC, 1 C, 5 M
P#4. Remainder	[26,36,37]	60 LOC, 1 C, 5 M
P#5. SyntheticProg1	[26]	65 LOC, 1 C, 5 M
P#6. SyntheticProg2	[26]	60 LOC, 1 C, 5 M
P#7. SyntheticProg3	[26]	62 LOC, 1 C, 5 M

– **RQ2**: How effective is the proposed technique in finding a reduced set of higher-order mutants?

4.1.3. Results and Discussion of Study #1

To answer the first research question **RQ1**, the *DataFolwBasedFOM* algorithm to find the first-order mutants was applied. According to the procedure of the empirical study, the *Muclipse* tool was run on the program to be mutated to generate *FOMs*. *Muclipse* generates 1114 mutated versions of the programs to be mutated. Table 8 presents the number of first-order mutants for each subject program, and the frequency of each mutation operator.

The analysis modulefinds a list of *def-c-use* pairs and *def-p-use* pairs for each subject program. The analysis module finds 124 *def-c-use* pairs for all subject programs and 196 *def-p-use* pairs which are reduced to 98 *def-p-use* pairs. The analysis module finds 320 *du-pairs* which are reduced to 222 *du-pairs* for all subject programs. Table 9 presents the number of *du-pairs* for each subject program. The mutant generation module generates 122 mutated versions of the programs to be mutated. This means that the proposed technique reduced 89% of the number of first-order mutants generated by *Muclipse* and presents the efficiency of data flow in aiding the reduction of the number of mutants. Figure 3 shows the number of *FOMs* generated by *Muclipse* and the proposed technique for each subject program.

To answer the second research question *RQ2*, the four techniques were applied: the *Muclipse* tool, the *LastToFirst* Algorithm, the *DifferentOperators* Algorithm, and the proposed *DataFolwBasedSOM* algorithm to generate all possible second-order mutants. Table 10 presents the number of second-order mutants (*SOMs*) generated by each one of these algorithms. Applying the *Muclipse* tool twice gives 186802 *SOMs*, which represents the worst case. The proposed algorithm generated 222 *SOMs*, while the *LastToFirst* algorithm generated 559 *SOMs*, and the *DifferentOperators* algorithm generated 595 *SOMs* for all subject programs. To compare the last three algorithms in reducing the number of higher-order mutants regarding *FOMs*, the reduction value of *FOMs* generated by *Muclipse* ($RR1 = (FOMs - SOMs)/FOMs$) was computed.

Table 8. Details of *FOMs* using Muclipse

#	Mutation Operators															Total
	AORB	AOIU	AODU	ROR	COD	SOR	LOI	ASRS	AORS	AOIS	AODS	COR	COI	LOR	LOD	
P#1	4	13	0	40	0	0	30	0	0	118	0	4	10	0	0	219
P#2	0	9	0	10	0	0	19	0	0	76	0	0	5	0	0	119
P#3	16	8	1	5	0	0	7	0	0	48	0	0	3	0	0	88
P#4	20	13	1	25	0	0	24	0	0	96	0	2	7	0	0	188
P#5	32	12	0	25	0	0	20	0	0	80	0	0	5	0	0	174
P#6	16	12	0	20	0	0	22	0	0	88	0	0	5	0	0	163
P#7	36	10	0	15	0	0	20	0	0	78	0	0	4	0	0	163
Total	124	77	2	140	0	0	142	0	0	584	0	6	39	0	0	1114

Table 9. The number of *du-pairs* for each subject program

# Subject program	dcu	dpu	Reduced dpu (Rdpu)	dcu+Rdpu	Mutation Points (mp)
P#1	16	72	36	52	22
P#2	10	20	10	20	14
P#3	14	10	5	19	13
P#4	24	30	15	39	21
P#5	20	20	10	30	18
P#6	22	28	14	36	19
P#7	18	16	8	26	15
Total	124	196	98	222	122

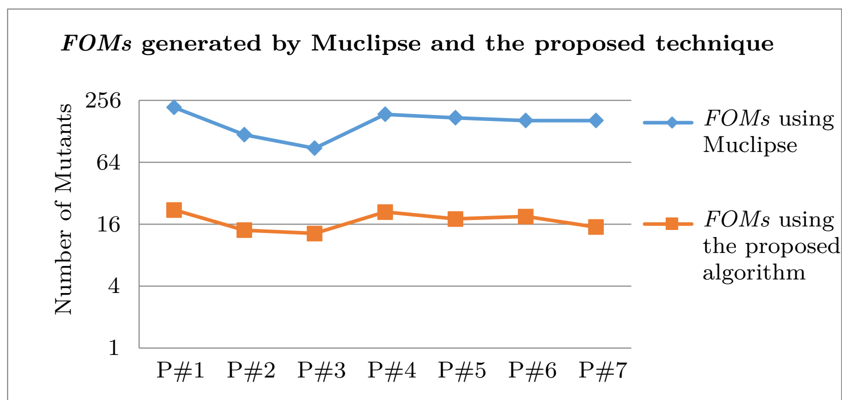

Figure 3. The number of *FOMs* generated by Muclipse and the proposed technique

Table 10 and Figure 4 show the reduction ratio of the 1114 first-order mutants generated by the *Muclipse* tool for each subject program. Our proposed algorithm reduced 88.07% of the 1114 *FOMs*, while the *LastToFirst* algorithm reduced 49.82%, and the *DifferentOperators* algorithm reduced 46.59% of 1114 *FOMs* for all subject programs. The results show that our proposed algorithm outperforms the *LastToFirst* algorithm by 38.25% and the *DifferentOperators* algorithm by 41.48%.

To compare the last three algorithms in reducing all possible number of higher-order mutants, the Authors computed the reduction value of *SOMs* generated by Muclipse ($RR2 = (MSOMs - ASOMs)/MSOMs$) where *MSOMs* is the number of second-order mutants generated by *Muclipse* tool, and *ASOMs* is the number of second-order mutants generated by one of the other three algorithms. Table 12 and Figure 5 show the reduction ratio *RR2* of the 186802 second-order mutants generated by the *Muclipse*

Table 10. The number of second-order mutants generated by the three algorithms

# Subject program	Muclipse	LastToFirst	DifferentOperators	The proposed algorithm
P#1	47557	110	118	52
P#2	13935	60	76	20
P#3	7634	44	48	19
P#4	35028	94	96	39
P#5	29995	87	87	30
P#6	26304	82	88	36
P#7	26349	82	82	26
Total	186802	559	595	222

Table 11. Reduction % of the number of first-order mutants generated by the three algorithms

# Subject program	LastToFirst	DifferentOperators	The proposed algorithm
P#1	49.77%	46.12%	76.26%
P#2	72.60%	65.30%	90.87%
P#3	79.91%	78.08%	91.32%
P#4	57.08%	56.16%	82.19%
P#5	60.27%	60.27%	86.30%
P#6	62.56%	59.82%	83.56%
P#7	62.56%	62.56%	88.13%
Total	49.82%	46.59%	80.07%

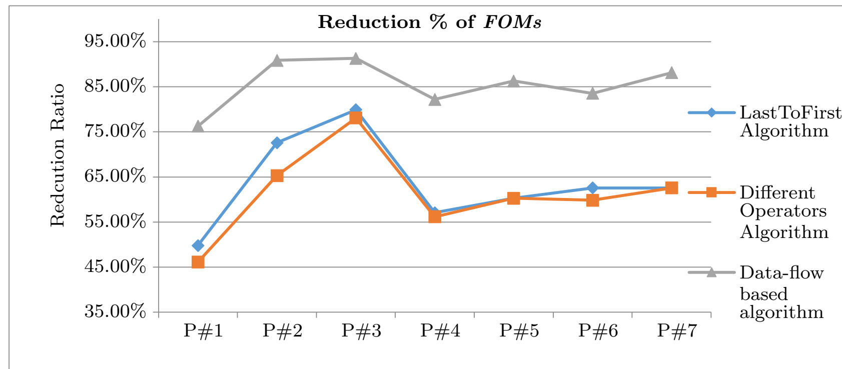

Figure 4. Reduction percentage of *FOMs* using the three algorithms

tool for all subject programs. Our proposed algorithm reduced 99.88% of all *SOMs* while the *LastToFirst* algorithm reduced 99.70% and the *DifferentOperators* algorithm reduced 99.68% of 186802 *SOMs* for all subject programs. The results show that our proposed algorithm outperforms the *LastToFirst* algorithm by 0.18%, and *DifferentOperators* algorithm by 0.20%. The results show the efficiency of data flow in aiding the reduction of the number of mutants, and the effectiveness of the proposed technique in finding a reduced set of higher-order mutants.

4.2. Empirical Study #2

4.2.1. Setup of Empirical Study #2

Subject Programs: The Authors selected four programs of the subject programs showed in Table 7 for conducting this empirical study to demonstrate that the proposed mutant generation technique does not lead to a substantial loss in the effectiveness of the mutation testing method. These programs are: *triangle*, *mid*, *power*, and *remainder*.

Table 12. Reduction % of the number of second-order mutants generated by the three algorithms

# Subject program	LastToFirst	DifferentOperators	The proposed algorithm
P#1	99.77%	99.75%	99.89%
P#2	99.57%	99.45%	99.86%
P#3	99.42%	99.37%	99.75%
P#4	99.73%	99.73%	99.89%
P#5	99.71%	99.71%	99.90%
P#6	99.69%	99.67%	99.86%
P#7	99.69%	99.69%	99.90%
Total	99.70%	99.68%	99.88%

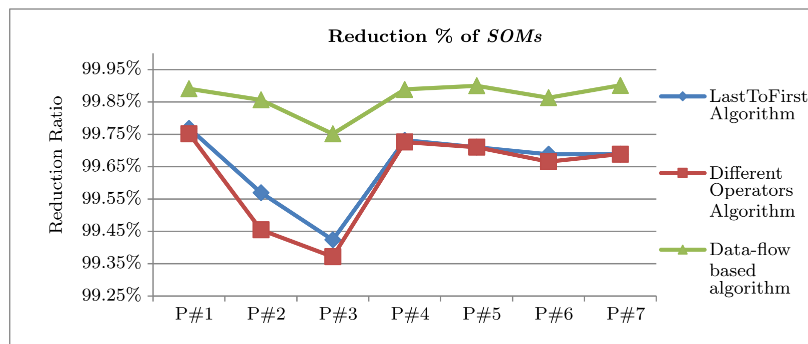

Figure 5. Reduction percentage of *SOMs* using the three algorithms

Procedure: the empirical study is conducted as follows.

1. The Authors randomly selected a set of the generated *FOMs* and *SOMs*. Table 13 shows the number and ratio of the selected *FOMs* and *HOMs*.
2. The Authors manually generated a set of test cases to kill all selected *FOMs*. Then, the selected *SOMs* were executed using this set of test cases.

Table 13. The number and ratio of selected *FOMs* and *HOMs*

#Subject program	*FOMs*	*SOMs*
P#1. Triangle	28 (12.8%)	12 (23.1%)
P#2. Mid	36 (30.3%)	10 (50.0%)
P#3. Power	63 (71.6%)	17 (89.5%)
P#4. Remainder	32 (17.0%)	10 (25.6%)
Total(Mean)	159 (25.9%)	49 (37.7%)

4.3. Objectives of Study #2

The Authors applied the study procedure to illustrate that the proposed mutant generation technique does not lead to a substantial loss in the effectiveness of the mutation testing method.

4.3.1. Results and Discussion of Study #2

To illustrate that the proposed mutant generation technique does not lead to a substantial loss in the effectiveness of the mutation testing method, the Authors selected approximately 26% of *FOMs* and 38% of *SOMs* as shown in Table 13. There is a difference between the ratio of *FOMs* and *SOMs* because most of the selected *SOMs* contained one of the selected *FOMs*. The Authors manually generated a set of test cases to kill selected set of *FOMs*. Table 14 shows the number of required test cases to kill the selected set of *FOMs*.

All *FOMs* and *SOMs* were selected using the generated test cases Then, we classified the selected *FOMs* and *SOMs* into killed, not killed, and equivalent (manually investigated) mutants. Table 15 and Table 16 show the classification, number, and ratio of *FOMs* and *SOMs*, respectively.

Table 14. The number of test cases

Subject program	P#1. Triangle	P#2. Mid	P#3. Power	P#4. Remainder	Total
No. of test cases	3	3	2	2	10

Table 15. Classification of the selected *FOMs*

Subject program	#Killed mutants(%)	#Not killed (%)	#Equivalent (%)	Total
P#1. Triangle	26 (93%)	2 (7%)	0 (0%)	28 (100%)
P#2. Mid	32 (89%)	0 (0%)	4 (11%)	36 (100%)
P#3. Power	52 (83%)	11 (17%)	0 (0%)	63 (100%)
P#4. Remainder	29 (91%)	0 (0%)	3 (9%)	32 (100%)
Total(Mean)	139 (89%)	13 (6%)	7 (5%)	159 (100%)

Table 16. Classification of the selected *SOMs*

Subject program	#Killed mutants(%)	#Not killed(%)	#Equivalent(%)	Total
P#1. Triangle	11 (92%)	1 (8%)	0 (0%)	12 (100%)
P#2. Mid	9 (90%)	0 (0%)	1 (10%)	10 (100%)
P#3. Power	14 (82%)	2 (12%)	1 6%)	17 (100%)
P#4. Remainder	10 (100%)	0(0%)	0(0%)	10 (100%)
Total(Mean)	44 (91%)	3 (5%)	2 (4%)	49 (100%)

The mutation score MS(P, T) was computed for each program using Eq. 1. Table 17 shows the mutation score for each program with respect to *FOMs* and *SOMs*. The mutation score shows that there is no significant loss in the efficiency of the generated mutants. The results of empirical study show that the proposed technique generated a smaller number of equivalent mutants.

Table 17. Mutation score of *FOMS* and *SOMs*

Subject Program	*FOM*	*SOM*
P#1. Triangle	92.9%	91.7%
P#2. Mid	100.0%	100.0%
P#3. Power	82.5%	87.5%
P#4. Remainder	100.0%	100.0%
Mean	93.8%	94.8%

4.4. Threats to Validity

- Construct Validity
  There are three important questions about the goal of the experiments. First, are the Authors measuring the construct they intended to measure? Although, the Authors intended to find a reduced set of higher-order mutants, some useless mutants (e.g., equivalent mutants and redundancies) can be generated and included in this set. Second, did the Authors translate these constructs correctly into observable measures? Although, the considered the def locations and use locations of the same variable, the mutations for other variables at these use locations are not considered. Third, did the used metrics have suitable discriminatory power? Although, the metric of the reduction is the ratio between the number of generated higher-order mutants to the number of all first-order mutants, it does not consider the subtlety of the generated higher-order mutants.
- External Validity
  The main external threat to validity; conditions that limit the ability to generalize the results of our empirical studies to a larger population of subjects programs, is the set of subject programs. Although the set of the subject programs contains some programs which have been used in many previous studies, the Authors cannot claim that these subjects represent a random selection over the population of programs as a whole. Although the set of the subject programs have been used

in many previous studies, a single researcher selected these programs which may influence results. Although, the Authors selected the subject programs in a neutral attitude, there is no guarantee that selection process was performed in unbiased way.

- Internal Validity
  There are some main internal threats to validity, which are the influences that can affect the dependent variables. First, although the mutation operators were selected in a significant way to prevent the generation of equivalent mutants, the equivalent mutants were not considered through the reduction ratio. Therefore, other empirical studies are required to overcome this problem. Second, although a common algorithm proposed by F. E. Allen and J. Cocke [31] was implemented to find the set of definition-uses, the accuracy of the implementation can influence the number of definition-uses pairs which have a strong effect on the number of generated mutants.

5. Related Work

Mutation testing has been developed by DeMillo et al. [3] and Hamlet [4] to create test data for killing the seeded mutations in the tested program [5]. The researchers classified mutants into two categories: 1) First-order mutants which are created by the injection of a unique fault in the tested program [5]; 2) Higher-order mutants which are produced by inserting two or more faults in the tested program [1]. Jia and Harman [38] provided a comprehensive analysis of trends and results of mutation testing techniques. This section reviews mutation operators design, generation of mutants, reduction of the number of mutants, and data flow analysis in mutation testing.

5.1. Mutation Operators and Mutant Generation

At the beginning of mutation testing, most mutation testing techniques targeted FORTRAN programs. Many mutation operators are presented for most of programming languages such as FORTRAN IV [39, 40], FORTRAN 77 [15, 41], Ada [42,43], ANSI C [13], and the Java programming language [44, 45]. Alexander et al. [46] presented a set of mutation operators to insert into Java utility libraries. Bradbury et al. [47] presented a set of mutation operators to the concurrent Java programs. Derezińska proposed a set of C# mutation operators [48, 49]. Ferrari et al. [50] suggested a set of mutation operators for Aspect-Oriented programs. Anbalagan and Xie [51, 52] presented a technique for creating mutants for pointcuts and detecting equivalent mutants.

5.2. Mutant Reduction

Considering all mutants makes mutation testing a computationally expensive technique. Therefore, reducing the number of the considered mutants without a significant loss of test effectiveness has become a key research problem. Suppose M is a set of mutants and T is a set of test data. The mutation score of the test set T applied to mutants M is $MS(M, T)$. Therefore, the mutant reduction problem is known as finding a subset of mutants m from M, where $MS(m, T) = MS(M, T)$. Offutt and Untch [53] classified mutant reduction techniques to three techniques. These techniques concentrate only on the *fewer*, the *faster*, or the *smarter* mutants. Jia and Harman [38] divided these techniques into two techniques. One technique reduces the created mutants and the other technique reduces the execution cost. There are four popular techniques to reduce the number of considered mutants: *mutant sampling*, *mutant clustering*, *selective mutation*, and *higher-order mutation*. In mutant sampling a percentage of mutants is randomly selected from the set of all mutants [10–12] and the remaining mutants are discarded [15, 54]. In mutant clustering [17] a subset of mutants is selected using clustering algorithms and the remaining mutants are discarded [55]. The Selective Mutation [56,57] can be achieved by reducing the number of applied mutation operators without a significant loss of test effectiveness [14].

Selective Mutation can be done by omitting two mutation operators [14], four mutation operators [53], or six mutation operators. Wong and Mathur selected mutation operators based on test effectiveness [58, 59]. Offutt et al. [60] classified Mothra mutation operators to three groups: statements, operands, and expressions and omitted operators from each class in turn. Mresa and Bottaci [61] considered mutants which have the ability to detect equivalent mutants. Jia and Harman [20, 22] suggested reducing the number of first-order mutants by replacing them with a single *HOM*. Langdon et al. have used genetic programming to generate higher-order mutants [62].

5.3. Data Flow Analysis and Mutation Testing

A number of work explored the role of data flow analysis in mutation testing. Girgis and Woodward [63] and Marshall et al. [64] studied applying data flow analysis in weak mutation testing. Offutt and Tewary [65] and Mathur and Wong [54] studied the coverage of mutation based and data flow criteria by each other. Wong and Mathur [66] compared the effectiveness of mutation and data flow testing in fault detection. A comprehensive comparison between mutation and data flow testing techniques based on findings reported in research articles can be found in [67]. This field is in need for a lot of work to study the role of data flow concepts in higher-order mutation testing. From the above discussion, it is clear that our work belongs to the mutant reduction category. In addition, it differs from all pervious mutant reduction work. It is the first work treating mutant reduction by reducing the locations of seeding mutation.

6. Conclusion and Future work

In this paper a new technique for generating a reduced set of higher-order mutants was introduced. The proposed technique uses data-flow concepts for the identification of the higher-order mutants. The generated set of higher-order mutants consists of a reduced number of mutants, which reduces the cost of higher-order mutation testing. In addition, the proposed technique can generate the higher-order mutants directly without generating the first-order mutants or by combining two or more first-order mutants. The results of the conducted experiments showed that the proposed technique outperforms the *LastToFirst* algorithm by 38.25%, and the *DifferentOperators* algorithm by 41.48% reducing the total possible number of higher-order mutants regarding *FOMs*. In addition, the proposed algorithm outperforms the *LastToFirst* algorithm by 0.18%, and the *DifferentOperators* algorithm by 0.20% in reducing all possible number of higher-order mutants. The obtained results showed the efficiency of data flow in aiding the reduction of the number of mutants and the effectiveness of the proposed technique in finding a reduced set of higher-order mutants. In future work, The Authors are planning to perform these studies with real and large subject programs. In addition, the future work will try to answer the questions: "Do data-flow based higher-order mutants create subtle faults?" and "What are the effects of the proposed approach in terms of overcoming realism and equivalent mutant problems of mutation testing?". Besides, the Authors will study the subsuming property of the generated mutants in the future work.

References

[1] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, Vol. 51, No. 10, Oct. 2009, pp. 1379–1393. [Online]. http://dx.doi.org/10.1016/j.infsof.2009.04.016

[2] W.B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *J. Syst. Softw.*, Vol. 83, No. 12, Dec. 2010, pp. 2416–2430. [Online]. http://dx.doi.org/10.1016/j.jss.2010.07.027

[3] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, Vol. 11, No. 4, Apr. 1978, pp. 34–41. [Online]. http://dx.doi.org/10.1109/C-M.1978.218136

[4] R.G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 4, July 1977, pp. 279–290.

[5] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1074–1081. [Online]. http://doi.acm.org/10.1145/1276958.1277172

[6] W.E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July 1982, pp. 371–379.

[7] Q.V. Nguyen and L. Madeyski, *Advanced Computational Methods for Knowledge Engineering: Proceedings of the 2nd International Conference on Computer Science, Applied Mathematics and Applications (ICCSAMA 2014)*. Cham: Springer International Publishing, 2014, ch. Problems of Mutation Testing and Higher Order Mutation Testing, pp. 157–172. [Online]. http://dx.doi.org/10.1007/978-3-319-06569-4_12

[8] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, April 2012, pp. 701–710.

[9] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, Vol. 40, No. 1, Jan 2014, pp. 23–42.

[10] A.T. Acree, "On mutation," Ph.D. dissertation, Georgia Inst of Tech Atlanta School of Information and Computer Science, Aug 1980.

[11] T.A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale Univ, 1980.

[12] I. Bluemke and K. Kulesza, *New Results in Dependability and Computer Systems: Proceedings of the 8th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, September 9-13, 2013, Brunów, Poland*. Heidelberg: Springer International Publishing, 2013, ch. Reduction of Computational Cost in Mutation Testing by Sampling Mutants, pp. 41–51. [Online]. http://dx.doi.org/10.1007/978-3-319-00945-2_4

[13] H. Agrawal, R.A. DeMillo, R. Hathaway, W.M. Hsu, W. Hsu, E. Krauser, R.J. Martin, A.P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Software Eng. Research Center, Computer Science Dept., Purdue Univ., Technical Report SERC-TR-41-P, 1989.

[14] A.P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Computer Software and Applications Conference, 1991. COMPSAC '91., Proceedings of the Fifteenth Annual International*, Sep 1991, pp. 604 – 605.

[15] K.N. King and A.J. Offutt, "A fortran language system for mutation-based software testing," *Softw. Pract. Exper.*, Vol. 21, No. 7, Jun. 1991, pp. 685–718. [Online]. http://dx.doi.org/10.1002/spe.4380210704

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

[17] S. Hussain, "Mutation clustering," Master's thesis, King's College London, 2008.

[18] A.J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, Vol. 1, No. 1, Jan. 1992, pp. 5–20. [Online]. http://doi.acm.org/10.1145/125489.125473

[19] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, Nov 2010, pp. 300–309.

[20] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Softw. Test. Verif. Reliab.*, Vol. 19, 2009, pp. 111–131.

[21] M. Harman, Y. Jia, and W.B. Langdon, "A manifesto for higher order mutation testing," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, April 2010, pp. 80–89.

[22] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Eighth IEEE International Working Conference onSource Code Analysis and Manipulation*, Sept 2008, pp. 249–258.

[23] P.M. Herman, "A data flow analysis approach to program testing," *Australian Computer Journal*, Vol. 8, No. 3, 1976, pp. 92–96.

[24] S. Rapps and E.J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, April 1985, pp. 367–375.

[25] P.G. Frankl and E.J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, Oct 1988, pp. 1483–1498.

[26] A.S. Ghiduk, M.J. Harrold, and M.R. Girgis, "Using genetic algorithms to aid test-data generation for data-flow coverage," in *Software En-*

gineering Conference, 2007. APSEC 2007. 14th Asia-Pacific, Dec 2007, pp. 41–48.

[27] I. Burnstein, *Practical Software Testing*, 1st ed. Springer Science+Business Media New York: Springer Professional Computing, 2003, ch. A Process-Oriented Approach.

[28] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, Vol. 19, No. 4, Dec. 2011, pp. 691–723. [Online]. http://dx.doi.org/10.1007/s11219-011-9142-y

[29] M.S. Hecht, *Flow Analysis of Computer Programs*. New York, NY, USA: Elsevier Science Inc., 1977.

[30] S. Rapps and E.J. Weyuker, "Data flow analysis techniques for test data selection," in *Proceedings of the 6th International Conference on Software Engineering*, ser. ICSE '82. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 272–278. [Online]. http://dl.acm.org/citation.cfm?id=800254.807769

[31] F.E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, Vol. 19, No. 3, Mar. 1976, p. 137. [Online]. http://doi.acm.org/10.1145/360018.360025

[32] B.H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical Softw. Engg.*, Vol. 14, No. 3, Jun. 2009, pp. 341–369. [Online]. http://dx.doi.org/10.1007/s10664-008-9083-7

[33] B.H. Smith and L. Williams, "An empirical evaluation of the muJava mutation operators," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, Sept 2007, pp. 193–202.

[34] Y.S. Ma and J. Offutt, "Description of method-level mutation operators for Java," 2005. [Online]. https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

[35] P. May, J. Timmis, and K. Mander, *Artificial Immune Systems: 6th International Conference, ICARIS 2007, Santos, Brazil, August 26-29, 2007. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ch. Immune and Evolutionary Approaches to Software Mutation Testing, pp. 336–347. [Online]. http://dx.doi.org/10.1007/978-3-540-73922-7_29

[36] C.C. Michael, G. McGraw, and M.A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, Vol. 27, No. 12, Dec. 2001, pp. 1085–1110. [Online]. http://dx.doi.org/10.1109/32.988709

[37] R.P. Pargas, M.J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Softw. Test., Verif. Reliab.*, Vol. 9, No. 4, 1999, pp. 263–282. [Online]. http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y

[38] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, Sept 2011, pp. 649–678.

[39] T.A. Budd, R.J. Lipton, R. DeMillo, and F. Sayward, "The design of a prototype mutation system for program testing," in *Proceedings NCC, AFIPS Conference Records*, 1978, pp. 623–627.

[40] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '80. New York, NY, USA: ACM, 1980, pp. 220–233. [Online]. http://doi.acm.org/10.1145/567446.567468

[41] A. J.Offutt and K.N. King, "A FORTRAN 77 interpreter for mutation analysis," *SIGPLAN Not.*, Vol. 22, No. 7, Jul. 1987, pp. 177–188. [Online]. http://doi.acm.org/10.1145/960114.29669

[42] J. Bowser, "Reference manual for Ada mutant operators," Georgia Inst. of Technology, Technical Report GIT-SERC-88/02, 1988.

[43] A.J. Offutta, J. Voas, and J. Payn, "Mutation operators for Ada," George Mason Univ., Technical Report ISSE-TR-96-09, 1996.

[44] S. Kim, J.A. Clark, and J.A. McDermid, "The rigorous generation of Java mutation operators using HAZOP," in *12th International Conf. Software and Systems Eng. and Their Applications*, 1999.

[45] Y.S. Ma, Y.R. Kwon, and J. Offutt, "Inter-class mutation operators for Java," in *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, 2002, pp. 352–363.

[46] R.T. Alexander, J.M. Bieman, S. Ghosh, and B. Ji, "Mutation of Java objects," in *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002, pp. 341–351.

[47] J.S. Bradbury, J.R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Mutation Analysis, 2006. Second Workshop on*, Nov 2006, p. 11.

[48] A. Derezińska, *Software Engineering Techniques: Design for Quality.* Boston, MA: Springer US, 2007, ch. Advanced mutation operators

applicable in C# programs, pp. 283–288. [Online]. http://dx.doi.org/10.1007/978-0-387-39388-9_27

[49] A. Derezinska, "Quality assessment of mutation operators dedicated for C# programs," in *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, Oct 2006, pp. 227–234.

[50] F.C. Ferrari, J.C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, April 2008, pp. 52–61.

[51] P. Anbalagan and T. Xie, "Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs," in *Proceedings of the Second Workshop on Mutation Analysis*, ser. MUTATION '06. Washington, DC, USA: IEEE Computer Society, 2006, p. 3. [Online]. http://dx.doi.org/10.1109/MUTATION.2006.3

[52] P. Anbalagan and T. Xie, "Automated generation of pointcut mutants for testing pointcuts in AspectJ programs," in *19th International Symposium on Software Reliability Engineering, 2008. ISSRE 2008.*, Nov 2008, pp. 239–248.

[53] A.J. Offutt and R.H. Untch, "Mutation testing for the new century," W.E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2001, ch. Mutation 2000: Uniting the Orthogonal, pp. 34–44. [Online]. http://dl.acm.org/citation.cfm?id=571305.571314

[54] A.P. Mathur and W.E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, Vol. 4, No. 1, 1994, pp. 9–31. [Online]. http://dx.doi.org/10.1002/stvr.4370040104

[55] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A novel method of mutation clustering based on domain analysis," in *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009*, 2009, pp. 422–425.

[56] A.S. Namin and J.H. Andrews, "Finding sufficient mutation operators via variable reduction," in *Proceedings of the Second Workshop on Mutation Analysis*, ser. MUTATION '06. Washington, DC, USA: IEEE Computer Society, 2006, p. 5. [Online]. http://dx.doi.org/10.1109/MUTATION.2006.7

[57] A.S. Namin and J.H. Andrews, "On sufficiency of mutants," in *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE COMPANION '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 73–74. [Online]. http://dx.doi.org/10.1109/ICSECOMPANION.2007.56

[58] W.E. Wong, "On mutation and data flow," Ph.D. dissertation, West Lafayette, IN, USA, 1993, uMI Order No. GAX94-20921.

[59] W.E. Wong and A.P. Mathur, "Reducing the cost of mutation testing: An empirical study," *J. Syst. Softw.*, Vol. 31, No. 3, Dec. 1995, pp. 185–196. [Online]. http://dx.doi.org/10.1016/0164-1212(94)00098-0

[60] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, Vol. 5, No. 2, Apr. 1996, pp. 99–118. [Online]. http://doi.acm.org/10.1145/227607.227610

[61] E.S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: an empirical study," *Software Testing, Verification and Reliability*, Vol. 9, No. 4, 1999, pp. 205–232. [Online]. http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<205::AID-STVR186>3.0.CO;2-X

[62] W.B. Langdon, M. Harman, and Y. Jia, "Multi objective higher order mutation testing with genetic programming," in *Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009. TAIC PART '09*, Sept 2009, pp. 21–29.

[63] M.R. Girgis and M.R. Woodward, "An integrated system for program testing using weak mutation and data flow analysis," in *Proceedings of the 8th International Conference on Software Engineering*, ser. ICSE '85. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 313–319. [Online]. http://dl.acm.org/citation.cfm?id=319568.319662

[64] A.C. Marshall, D. Hedley, I.J. Riddell, and M.A. Hennell, "Static dataflow-aided weak mutation analysis (sdawm)," *Inf. Softw. Technol.*, Vol. 32, No. 1, Jan. 1990, pp. 99–104. [Online]. http://dx.doi.org/10.1016/0950-5849(90)90053-T

[65] A.J. Offutt and K. Tewary, "Empirical comparisons of data flow and mutation testing," 1992.

[66] W.E. Wong and A.P. Mathur, "Fault detection effectiveness of mutation and data flow testing," *Software Quality Journal*, Vol. 4, No. 1, pp. 69–83. [Online]. http://dx.doi.org/10.1007/BF00404650

[67] S. Kakarla, S. Momotaz, and A.S. Namin, "An evaluation of mutation and data-flow testing: A meta-analysis," in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, March 2011, pp. 366–375.