Journal of Theoretical and Applied Information Technology
15th August 2019. Vol.97. No 15
© 2005 – ongoing JATIT & LLS

ISSN: **1992-8645** www.jatit.org E-ISSN: **1817-3195**

AN EMPIRICAL EVALUATION OF THE SUBTLETY OF THE DATA-FLOW BASED HIGHER-ORDER MUTANTS

AHMED S. GHIDUK[1,2], AHMOUD ROKAYA[1,3]
[1]College of Computers and Information Technology, Taif University, Saudi Arabia
[2]Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Egypt
[3]Department of Mathematics (Computer Science), Faculty of Science, Tanta University, Egypt
E-mail:{[1,2]asaghiduk, [1,3]mahmoudrokaya}@tu.edu.sa

ABSTRACT

Mutants are erroneous forms of the source code generated by deliberately inserting one fault (first-order mutant) or more (higher-order mutant) into the source code. Smart mutants that require numerous numbers of test cases to be killed are called Subtle Mutants (SMs). These mutants are required in order to increase the efficiency and effectiveness of test cases. Creation of these mutants is an expensive step especially in higher-order mutation testing. Data-flow analysis has been effectively applied to create higher-order mutants and overcome the explosion problem. To the best of our knowledge, subtle mutant generation with the aid of data-flow concepts and identifying them among all mutants have not been studied adequately. In this paper, an empirical study to evaluate the impact of data-flow analysis on the subtlety of higher-order mutants is introduced. Therefore, this study discusses two research questions: which mutants are more subtle data-flow based second order mutants (DFSOMs) or their constitute FOMs mutants? And which mutants are harder to be killed or covered DFSOMs or all du-pairs criterion? The results of the conducted experiments showed that the subtlety of data-flow based second order mutants (DFSOM) is higher than their constitute first-order mutants by 6% in average. In addition, DFSOM criterion dominates all du-pairs criterion and covering (killing) DFSOM criterion is harder than covering all du-pairs criterion by 14.6% in average.

Keywords: *Mutation Testing, Higher-Order Mutants, Subtlety Mutants, Data-flow analysis*.

1. INTRODUCTION

Mutation testing concept has been initialized by DeMillo et al. [**1**] and Hamlet [**2**]. Techniques of mutation testing can be applied for measuring the effectiveness of any test suite, simulating any test criterion, and finding the required test inputs [**3**], [**4**]. To measure the quality of a test suite, mutants are run against the test suite. Then, the quality of the test suite can be estimated by the mutation score which is the ratio of mutants recognized by this test suite. For simulating a test criterion, mutants can be created by seeding faults at specific positions. For generating test data, inputs can be created for killing the mutants.

Recently, mutation testing has been classified into two major methodologies. The first methodology is the traditional or the original one suggested by DeMillo et al. [**1**] and Hamlet [**2**]. This methodology concentrates on applying mutation testing concepts on first-order mutants (FOMs) which are constructed by placing single fault in the source code [**5**], [**6**], [**7**]. The second methodology is higher-order mutation testing advocated by Jia and Harman [**8**]. This methodology is generalization of the traditional one and concentrates on higher-order mutants (HOMs) which are created by injecting the source code by two or more faults [**8**], [**9**]. Smart mutants that require numerous numbers of test cases to be killed are called Subtle Mutants (SMs) [**10**]. These mutants are required to increase the efficiency and effectiveness of test cases. Creation of these mutants is an expensive phase especially in higher-order mutation testing.

Although, there are a lot of techniques to construct the FOMs and HOMs and reduce their number, a very few number of techniques have been presented to construct SMs.

Acree [**5**] and Budd [**6**] applied mutant sampling method to decrease the number of mutants. Agrawal et al. [**11**] and Mathur [**12**] reduced the number of mutants by reducing the number of mutation operators. Offutt et al. [**7**] and [**13**] used selective mutation approach which elects a small set of operators to produce a subset of all potential mutants without losing test effectiveness. Husain [**14**] employed clustering algorithms to pick out a subset of mutants. Second-order mutation testing [**15**], [**16**], [**17**], [**18**], [**19**], [**20**] and [**21**] in particular, and higher-order mutation testing [**8**], [**22**], [**3**], and [**10**] in general, have been successfully applied to reduce the number of mutants [**9**].

Jia and Harman [**8**] and [**10**] presented the conceptions of subsuming higher-order mutants. They claimed that the subtle higher-order mutants can be considered as the subsuming higher-order mutants, which are harder to be killed than the first-order mutants from which they are composed [**10**]. In addition, they presented the conceptions of strong subsuming higher-order mutants, which can only be killed using a subgroup of the overlapped test cases that kill each first-order mutants from which they are composed [**10**]. They presented a search based methodology for identifying the subsuming higher-order mutants [**10**]. Jia and Harman introduced a measure in terms of the number of test cases for the subtlety of both first and higher-order mutants. By definition, this measure is the quotient of the ratio of number of test cases that kill HOMs out of the total number of test cases and the ratio of the number of test cases that kill FOMs out of the total number of test cases [**10**]. If this measure is greater than 1, this means that the higher-order mutant is weaker than the first-order mutants from which it is composed. If this measure is 0, this means that the HOM is a potential equivalent HOM. From 1 to 0, the higher-order mutant turns out gradually to be stronger than the first-order mutants from which it is composed. Langodn et al. [**3**] employed genetic programing for finding set of hard to kill higher-order mutants. They proposed a multi-objective fitness function based on the semantic and syntactic distances. The semantic distance is defined as the number of test cases which cause a mutant and original program act in a diverse manner. The syntactic distance is defined as total number of changes in the logical control structure. In addition, Harman et al. [**23**] studied the potential enhancement in the efficiency and effectiveness of the test due to using the strongly subsuming higher-order mutants.

Omar and Ghosh [**24**] suggested four approaches for generating higher-order mutants in AspectJ applications. They assessed the proposed approaches in terms of their power to improve test effectiveness by generating hard to kill mutants and their power to reduce test effort by reducing the number of mutants compared to first-order mutants. Omar et al. [**25**], [**26**] and [**27**] proposed set of search guided approaches to create subtle higher-order mutants. They proposed a new fitness function to evaluate the subtlety of the mutants. This function uses two metrics: fault detection difference between the higher-order mutant and its constitute first-order mutants, and difficulty of killing this higher-order mutant.

Nguyen and Madeyski [**28**] discussed some mutation testing problems and reviewed the approaches for constructing the good higher-order mutants. They continued their work and proposed in [**29**], [**30**], [**31**], [**32**], [**33**] a multi-objective optimization algorithm to create valuable higher-order mutants. Nguyen [**34**] compared the subtlety of higher-order and first-order mutants. They demonstrated that half of all generated higher-order mutants are harder to kill than its constituent first-order mutants.

Abuljadayel and Wedyan [**35**] proposed an approach for generating higher-order mutants and reducing the number of equivalent mutants. The proposed approach employed genetic algorithm to find the hard to kill higher-order mutants.

In earlier work, Ghiduk [**21**] employed the concepts of data flow analysis to reduce the number of higher-order mutants by electing specific locations in the tested program to be mutated. Ghiduk's approach mutates only the locations of *def* points and *use* points to construct the mutants leading to reduce their number. In this method, a second-order mutant can be constructed by seeding two mutations one at the *def* point and the second mutation at the *use* point of the same *def-use* pairs. Besides, Kintis and Malevris [**36**] employed these concepts to detect equivalent mutants. Recently, Ghiduk et al. [**37**] introduced a systematic literature review for higher-order mutation testing techniques. All higher-order mutation testing issues and all approaches which handled these issues are discussed by that work. To the best of our knowledge, data flow analysis concepts [**38**], [**39**] have been effectively used in many software testing aspects specially test data generation [**40**], [**41**]. Although Ghiduk [**21**] introduced an approach based on data flow for generating the higher-order mutants and reducing their number, the data flow concepts have never been applied for finding the subtle higher-order mutants.

The main contribution of this paper is conducting an empirical evaluation of the subtlety of the class of higher-order mutants which are generated by the aid of data flow. This empirical study considers the following research questions:

RQ1: Which mutants are more subtle DFSOMs or their constitute FOMs mutants?

RQ2: Which mutants are harder to be killed or covered DFSOMs or all du-pairs?

The remainder of this paper is structured as follows. Some important basic concepts and definitions are given in Section 2. Section 3 presents brief description for generating data flow based higher-order mutants. The details of this empirical study and its results are presented in Section 4. The previously published research and the related work are presented in Section 5. Section 6 introduces the conclusion of this paper and the future work.

2. BACKGROUND

In this section, some basic concepts that will be used throughout this work are presented.

2.1 Mutation testing

Mutation testing [**1**], [**2**], [**5**] and [**6**] needs three necessary inputs: the tested program, the mutation operators, and test suite. Mutation testing is performed according to the following procedure. Firstly, the tested program is executed against the test suite to verify its correctness. If it holds faults, it must be corrected in advance before continuing the mutation testing procedure. Secondly, a class of mutants is created by seeding faults into the tested program using the mutation operators. A mutant is formed by creating one or more minor change into the source program. Thirdly, all mutants and the tested program will be executed against the test suite. If the outputs of executing a mutant are not the same outputs of executing the tested program for any test case in the test suite, this mutant is called "*killed mutant*" otherwise it is called "*alive mutant*". *Alive mutant* can be *killable* mutant or *equivalent* one which has similar behavior as the tested program and needs extra human work to kill it [**16**]. The quality of the test suite can be evaluated by the *mutation score* (MS) formula [**42**].

$$MS = \frac{\#\ of\ killed\ Mutants}{Total\ no.\ of\ Mutants - no.of\ Equivalent\ Mutants} \quad (\boldsymbol{Eq.}\,1)$$

For instance, Table 1 presents a source code segment p, two first-order mutants FOM_1 and FOM_2 and a second-order mutant SOM formed by merging FOM_1 and FOM_2. FOM_1 is constructed by replacing the "!=" operator in the source code p with the "<" operator in the mutated code p'. FOM_2 is constructed by replacing the "= =" operator in the source code p with the ">" operator in the mutated code p'. In addition, Table 1 presents a test input which kills FOM_2 and SOM but it cannot kill FOM_1.

Despite of the effectiveness of mutation testing in assessing the quality of the test suite, it has three main weaknesses. These weaknesses are the massive number of mutants, equivalent mutant, and realism problem [**9**]. A great number of mutants can be formed during the mutant generation stage even for trivial programs. The code segment p can be mutated into at least 12 first-order mutants by applying only the six relational operators (= =, !=, >, >=, <, and <=) and the three conditional operators (&&, || and ^). Therefore, the execution of mutants (third step of mutation testing) is very costly. For example, if the segment code p has 100 test cases, it needs (1+12)*100 = 1300 executions [**9**]. To minimize the execution cost, Howden [**43**] suggested weak mutation [**44**] in which result of mutant is checked immediately after executing the mutated component to see if the mutant is killed or not. Besides, mutants don't represent realistic faults due to they are formed by simple syntactic changes but 90% of real faults are complex [**3**]. Subtle mutants can help in overcoming this problem [**3**]. Furthermore, many mutation operators can produce equivalent mutants [**16**].

Table 1: An Example of Mutation Operation

Tested Code p	Mutated Code p'		
	FOM_1	FOM_2	SOM
if (m != 0 && n = = 0)	if (m < 0 && n = = 0)	if (m != 0 && n > 0)	if (m < 0 && n > 0)
Test input: m = -5, n = 0 Output of p is: true	Output of p' is: true alive mutant	Output of p' is: false killed mutant	Output of p' is: false killed mutant

2.2 Higher-Order Mutation Testing

Higher-order mutation testing (HOMT) is considered an expansion of classical mutation testing. Therefore, HOMT is performed using the same procedure of the classical mutation testing given in subsection 2.1. Higher-order mutants are built by inserting two or more mutations into the source code or by merging two or more first-order mutants [**8**].

Recently, higher-order mutants are divided into two main categories: subtle mutants and naive mutants. Subtle mutants are those hard to kill (i.e. those mutants that need wide range of test cases to be killed). Naive mutants are those easy to kill (i.e. those mutants that can be killed by most of the test cases). Jia and Harman [**8**] categorized higher-order mutants to six types based on the way that they are coupled or subsuming. Coupled higher-order mutants are those mutants that are coupled to first-order mutants. Subsuming higher-order mutants are those mutants that their constituent mutants partly mask one another. In formal, these six types are defined as follows. Suppose that h is a higher-order mutant formed from n first-order mutants (f_i, i = 1... n) and T is the current test suite. Let $T_h \subset T$ is the set of test cases which kills h and $T_i \subset T$ is the set of test cases that kills the constituent first-order mutant f_i . The mutant h is: strongly subsuming and coupled if $T_h \subset \bigcap_i T_i$ and $T_h \neq \emptyset$; weakly subsuming and coupled if $|T_h| < |\bigcup_i T_i|, T_h \neq \emptyset$ and $T_h \cap \bigcup_i T_i \neq \emptyset$; weakly subsuming and decoupled if $|T_h| < |\bigcup_i T_i|, T_h \neq \emptyset$ and $T_h \cap \bigcup_i T_i = \emptyset$; non-subsuming and decoupled if $|T_h| \geq |\bigcup_i T_i|, T_h \neq \emptyset$ and $T_h \cap \bigcup_i T_i \neq \emptyset$; non-subsuming and decoupled if $T_h = \phi$ (equivalent); non-subsuming and coupled if $|T_h| \geq |\bigcup_i T_i|$(useless).

2.3 Data Flow Analysis

The structure of any program can be modeled graphically by the control flow graph. A control flow graph comprises of a group of nodes and a group of edges. Each node represents a statement of the program code and each edge is an ordered pair of two adjacent nodes. A path is a series of nodes, from the entry node to the exit one, connected by edges [**45**] and [**46**].

Instead of the logic or control structure of the program data flow testing is centered on the role of variables (data) in the code [**38**]. Therefore, data flow analysis focuses on finding all "Definition-Use Associations (*dua*)" for each variable *x* in the tested program. Each *dua* consists of a triple (*x*, *d*, *u*) in which *d* is a statement holding a *definition* of *x* and *u* is a statement holding a *use* of *x* that can be reached by *d* through some paths [**39**], [**40**]. If the value of a variable *x* is assigned or changed in a statement, this operation is called a *definition* (*def*) of *x*. In addition, if the value of the variable *x* is used in a statement and not changed, this operation is called a *use* of *x*. If the *use* is located in a *predicate*, it is called *p-use*. Besides, if the *use* is located in a computation statement, it is called *c-use*.

3. DATA FLOW BASED HIGHER-ORDER MUTANT

This section describes briefly our early work [**21**] for constructing higher-order mutants. In this work, using the data flow concepts we presented a technique [**47**] for constructing higher-order mutants and decreasing their number through decreasing the number of mutated positions in the tested program. Figure 1 presents the algorithm of this technique. This technique has two main phases which can be summarized as follows.

Data flow analysis phase: This phase applied the method presented by Allen and Cocke [**47**] on the tested Java program to find all *dua* in it.

Second-order mutant generation phase: This phase uses all *dua* to create the higher-order mutants. It considers the locations of *def* points and the locations of *use* points as locations to be mutated. For producing a second-order mutant, two mutation operators are seeded into the tested program such that the first mutation is seeded at the *def* site and the second mutation is seeded at the *use* site and the two sites belong to the same *dua*. Therefore, this technique requires three main inputs (*dua* positions, mutation operators, and the tested program) to create higher-order mutants without requiring the first-order mutants. To perform its task, this technique uses two main functions. The first function is Operator.select() which applies various methods to choice two mutation operators to seed them into the original code. The used operator selection methods are: 1) *not selected yet* which selects two operators that were not selected so far; 2) *different operator* which selects two different operators; and 3) *different category* which selects each operator from one category such as arithmetic, relational, conditional, and logical categories. The second function is allDefUsePairs.select() which selects one *def-use pairs* to be mutated. For creating mutants of order greater than the second order, the algorithm given in Figure 1 is applied number of times more than one time with exchanging the input program to be the output program of the preceding cycle.

```
Algorithm DataFolwBasedSOM(program,
allDefUsePairs[], operators[])
let secondOrderMutants =Ø
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);
    end while
end while
return secondOrderMutants;
```

Figure 1: Algorithm for generating second-order mutants.

4. THE EMPIRICAL EVALUATION PROCEDURE

4.1 Empirical Studies Guidelines

Scholars introduced a preliminary set of research guidelines for designing, conducting and evaluating empirical studies [48], [49]. Kitchenham et al. [49] introduced guidelines for the key areas: "experimental context", "experimental design", "conduct of the experiment and data collection", "analysis", "presentation of results", and "interpretation of results". In this empirical evaluation research, the procedure proposed by Kitchenham et al. [49] and the guidelines which are applicable for this empirical study are followed. These guidelines are collected and summarized in Table 8.

The following sections present how our study is performed according to these guidelines.

4.2 Experimental Context

Although standard contextual information aids in comparing the related studies or replicating them and understanding tools and techniques, unfortunately software engineering doesn't have definite standards for deciding which contextual information would be involved in the "study design", gathered throughout the study, and reported in the results [49].

According to Table 8, our empirical study discussed the similar studies in section 2. In addition, the hypotheses of this study are:

H1: Higher-order mutants can be generated by aiding of data flow analysis concepts.

H2: Data flow analysis concepts have the ability to reduce the number of higher-order mutants.

The addressed research questions of this study are:

RQ1: Which mutants are more subtle DFSOMs or their constitute FOMs mutants?;

RQ2: Which mutants are harder to be killed or covered DFSOMs or all du-pairs?

4.3 Experimental design

The population from which the subjects are selected is Java programs which have been used in the previous similar researches. To conduct our empirical study, set of Java programs from the earlier researches has been selected. These programs include benchmarks such as Mid, Remainder, Triangle, and Power, and some artificial programs of diverse configurations and structures. Table 2 presents the specifications of these subjects: the column title of subject program provides code and title for each subject; the column reference presents some of the earlier researches which utilized these subjects; and the column scale introduces the specifications of subject.

This study considered only the programs which contains only on class and any number of methods of any size. Each subject program is treated separately of the other subjects. Therefore, the procedure of the study is applied on each subject program and the selected metric is computed for each subject.

Table 2: The specifications subjects.

# Title of subject program	Reference	Scale (#LOC, #Classes, #Methods
SP#1. Triangle	[3], [50], [19], [41]	73 LOC. 1 C, 6 M
SP#2. Mid	[19], [41], [51]	61 LOC, 1 C, 6 M
SP#3. Power	[41], [52], [51]	49 LOC, 1 C, 5 M
SP#4. Remainder	[41], [52], [51]	60 LOC, 1 C, 5 M
SP#5. Synthetic1	[41]	65 LOC, 1 C, 5 M
SP#6. Synthetic2	[41]	60 LOC, 1 C, 5 M
SP#7. Synthetic3	[41]	62 LOC, 1 C, 5 M

4.4 Conduct of the Experiment and Data Collection

A Java based tool has implemented to automatically perform the empirical studies. The stages of the tool and the procedure of the empirical studies are as follows.

1. Input step: Get the subject (Java program).
2. Data flow analysis: Apply the data flow analysis procedure to find the set of all *dua*.
3. Mutant construction: Get the set of mutation operators. The study used the set of method-level operators proposed by Y. Ma and J. Offutt [53]. The study applied the arithmetic (AORB, AORU, AORS, AOIS, AOIU, AODS, AODU), relational (ROR), conditional (COR, COI, and COD), and logical (LOR, LOI, and LOD) operators. The study selected these operators because they are the most repeated in the subject programs. Then, generate second-order mutants as much as possible using the algorithm given in Figure 1. After that, discard any redundancy or equivalent mutants.
4. Test suite generation: Create a suitable test suite. The study generates this test suite using a GA-based tool developed by the first author (STDGenGA) [54]. GA has been used successfully to generate test data [55] [56] [54] [41].
5. Test suite execution: Execute the subject and its mutants against the test suite. Then, compute the

subtlety of the mutants for the subject using the metrics discussed in section 2.3.

4.5 Results Presentation, Analysis and Interpretation

In this section, the results of applying each step of the procedure of this study are presented and then these results are discussed in details. The Java function *Midnum* (given in **Error! Reference source not found.**) is selected from the subject program *Mid* to illustrate the steps of the procedure.

In the second step of the procedure, data flow analysis step, the tool applies the data flow analysis concepts to find for the tested program (which input in first step) all data actions (*defs* and *uses*) for each variable. Then, by coupling these *defs* and *uses* the tool finds all *dua* for each variable. For instance, in this step of the procedure 20 *dua* were generated for the Java function presented in **Error! Reference source not found.**. **Error! Reference source not found.** presents the data flow actions for the variables: *x*, *y*, *z*, *mid* and 20 *dua* for these variables. In this stage, a set of *dua* is created for each subject. Table 3 presents for each subject the number of *dua*. The tool created 222 *dua* for all subjects.

Table 3: No. of dua for each subject.

# Subject Prog	SP#1	SP#2	SP#3	SP#4	SP#5	SP#6	SP#7	Total
Dua	52	20	19	39	30	36	26	222

In the third step of the procedure, mutant construction step, the tool applies the algorithm given in Figure 1 to construct set of second-order mutants for each subject. In this step, the algorithm gets as inputs the tested program and its *dua* and the set of operators illustrated in Table 9. Then according to the type of the statements that are assigned by the *def* and *use* in the current *dua*, it generates all possible second-order mutants. For the Java function presented in **Error! Reference source not found.**, the tool reads this function, the 20 *dua* and the operators and their possible operations. For instance, suppose the current *dua* is (x, 3, 8). Subsequently, the *def* statement is an assignment statement and the *use* statement is an *if* statement. This assignment statement number 3 (x = num1;) can be mutated only by the insertion of one of arithmetic unary (+, -) or short-cut (op++, ++op, op --, -- op) operators. Similarly, the *if* statement number 8 (if (x < Y) ;) can be mutated by replacing the relational operator '<' by one of the relational operators (>, >=, <=, = =, !=). Consequently, the mutation of these two statements individually can construct 11 first-order mutants such as x = ++ num1; x = - num1; if (x > y); ... etc. In addition, the mutation of these two statements can construct from 5 to 30 second-order mutants according to the selection approach of the operators. The "*not selected yet*" approach can construct 5 second-order mutants but the "*all permutations*" can construct 30 second-order mutants. In all cases the set of the 11 first-order mutant is the constituent of these second-order mutants. The "*not selected yet*" approach can construct 110 second-order mutants constructed of 70 constituents first-order mutants but the "*all permutations*" can construct 660 second-order mutants of 230 constituents first-order mutants. Table 4 presents the number of second-order mutants and their constituents of the example Java function presented in **Error! Reference source not found.** according to *dua* of each variable in this function. Table 10 presents the number of second-order mutants for each subject programs and the number of constituents. There are 13265 second-order mutants construed by "*all permutations*" for all subjects of 1250 constituents first-order mutants and 1315 second-order mutants construed by "*not selected yet*" of 775 constituents first-order mutants. The set of mutants generated by the tool using "*not selected yet*" method will be used throughout the empirical study to evaluate its subtlety. All equivalent mutants and stillborn ones are discarded from consideration in the next step.

Table 4: No. of second-order mutants for the example Java function and their constituent.

Dua	No. mutation at *def*	No. mutation at *use*	All permutations		Not selected yet	
			SOM	Constituents	SOM	Constituents
dua of *x*	36	32	192	68	32	64
dua of *y*	30	27	162	57	27	54
dua of *z*	24	21	126	45	21	42
dua of *mid*	30	30	180	60	30	60
Total	120	110	660	230	110	220*

* There is duplication in this number because the constituents are counted according to the *dua*. This number is 70 without duplication.

In the fourth step of the procedure, test suite generation step, a genetic algorithm technique (STDGenGA) [**54**] was used by the tool to generate a test suite to cover all *dua* for each subject program. Table 5 presents the number of the generated test cases for each subject program.

Table 5: Number of test cases.

Subject	SP#1	SP#2	SP#3	SP#4	SP#5	SP#6	SP#7	Total

program								
No. of test cases	14	10	6	9	5	8	9	61

In the last step (step 5) of the procedure, test suite execution, each subject program and its non-equivalent and non-stillborn FOMs and SOMs mutants are executed against the generated test suite. For each program the number of killed and alive FOMs and SOMs mutants are counted and the coverage ratio of data flow criterion (all du-pairs) as well.

Table 6: Subtlety of DFSOM and Data flow criteria against the same test suite for each subject program.

Subject program	DFSOM criterion		Data flow criterion	
	Killed	Alive	Covered (killed)	Not Covered (alive)
SP#1	65%	35%	100%	0%
SP#2	90%	10%	100%	0%
SP#3	64%	36%	93%	7%
SP#4	80%	20%	83%	17%
SP#5	70%	30%	90%	10%
SP#6	90%	10%	96%	4%
SP#7	100%	0%	100%	0%
Average	79.9%	20.1%	94.6%	5.4%

Table 6 presents for each subject program the ratios of killed and alive data-flow based second order mutants (DFSOM) and the ratios of covered and not-covered du-pairs. The results given in Table 6 showed that data-flow based second order mutants (DFSOM) criterion is subtle than all du-pairs criterion where the ratio of alive DFSOM for all subject programs is 20.1% in average and the ratio of not covered du-pairs is 5.4% in average although the test suite is generated to cover all du-pairs. Form the results given in Table 6 and Figure 2, we concluded that killing all DFSOM guarantee covering all du-pairs (i.e., DFSOM criterion dominates Data flow criterion). Therefore, DFSOMs is harder to be killed or covered than all du-pairs (this answers RQ2).

Table 7 presents for each subject program the ratios of killed and alive data-flow based second order mutants (DFSOM) and the ratios of killed and alive of their constitute FOMs. The results presented in Table 7 showed that data-flow based second order mutants (DFSOM) criterion is subtle than their constitute FOMs where the ratio of alive DFSOM for all subject programs is 20% in average and the ratio of alive FOMs is 14% in average. Form the results given in Table 7 and Figure 3, we concluded that DFSOMs mutants is more subtle than their constitute FOMs (this answers RQ1).

4.4 Threats to Validity

4.4.1 External validity

The main external threat to validity is the set of subject programs. Although the subject programs have been utilized in many previous studies, we cannot claim the programs are a random collection of the population of programs as a whole which may influence results.

4.4.2 Internal validity

The main internal threats to validity is the generation of equivalent mutants and stillborn ones, although we didn't consider these mutants through the test execution step by discarded these kind of mutants manually but this process is time consuming process and may be inaccurate.

Table 7: Subtlety of DFSOM and their Constitute FOM against the same test suite for each subject program.

Subject program	DFSOM		Constitute FOM	
	Killed	Alive	Killed	Alive
SP#1	65%	35%	79%	21%
SP#2	90%	10%	100%	0%
SP#3	64%	36%	80%	20%
SP#4	80%	20%	80%	20%
SP#5	70%	30%	70%	30%
SP#6	90%	10%	90%	10%
SP#7	100%	0%	100%	0%
Average	80%	20%	86%	14%

5. RELATED WORK

Up to now, the researchers [**8**], [**27**], [**29**], [**35**] employed only some search based techniques such as genetic algorithm, local search, greedy algorithm, and hill climbing algorithm to construct higher-order subtle mutants. Therefore, there are many metrics to evaluate the subtlety of higher-order mutants.

Jia and Harman [**8**] presented a measure to find the fragility of each of the first and higher-order mutants. They defined the fragility of mutants as the ratio between the number of test cases which kill these mutants and the total number of test cases in the test suite. Therefore, the value of fragility changes gradually from zero to one, while the mutant changes from equivalent to the weakest. Then, they introduced a metric for measuring the hardness of the mutants as the ratio between the set of higher-order mutants and their constituent first-order mutants. The value of this metric is greater than or equal zero. The zero-valued mutants are potential equivalent higher-order mutants. As the value of this metric decreases from one to zero, the hardness of the higher-order mutants increases gradually than their constituent first-order mutants. If the value of this metric is greater than one, the higher-order mutants are weaker than their constituent first-order mutants.

Nguyen and Madeyski [**29**] suggested three objective functions (Φ_1, Φ_2, and Φ_3) and one fitness function (F), which are used together to assess the higher-order mutants and identify the subtle ones. These four functions can be described as follows. Suppose that T is the set of all test cases, $T_{F1} \subset T$ is the set of test cases which kill the first-order mutant FOM_1, $T_{F2} \subset T$ is the set of test cases which kill the first-order mutant FOM_2, $T_H \subset T$ is the set of test cases which kill the higher-order mutant HOM created from FOM_1 and FOM_2. By definition $\Phi_1 = \frac{|T_H \cap T_{F1} \cap T_{F2}|}{|T_H|}$, $\Phi_2 = \frac{|T_H - (T_{F1} \cup T_{F2})|}{|T_H|}$, $\Phi_3 = \frac{|(T_H \cap (T_{F1} \cup T_{F2})) - (T_{F1} \cap T_{F2})|}{|T_H|}$, and $F(H) = \frac{|T_H|}{|T_{F1} \cup T_{F2}|}$. The values of Φ_1, Φ_2, Φ_3, and F lie between 0 and 1. According to these definitions, Nguyen and Madeyski showed that the subtle higher-order mutants are those mutants with $0 < \Phi_1 \leq 1$, $\Phi_2 = 0$, $\Phi_3 = 0$, and $F \leq 1$.

Omar and Ghosh [**27**] combined two metrics to evaluate the subtlety of higher-order mutants. The first metric, μ_1, compares between the fault detection effectiveness of the higher-order mutant and its constituent first-order mutants. The metric (μ_1) is the ratio of the difference between the cardinal number of the union set (U) of all test cases which kill the higher-order mutant or its constituent first-order mutants and the cardinal number of their intersection set ($\cap$) out of the cardinal number of the union set (U) (i.e., $\mu_1 = \frac{|U| - |\cap|}{|U|}$). The second metric, μ_2, measures the hardness of killing the higher-order mutant. The metric (μ_2) is the ratio of the difference between the cardinal number of the union set (U) and the cardinal number of the set (T) of test cases which kill the higher-order mutant out of the cardinal number of the union set (U) (i.e., $\mu_2 = \frac{|U| - |T|}{|U|}$). Then, they combined μ_1 and μ_2 into a single metric F to find the fitness value of higher-order mutant using the formula $F = \alpha \mu_1 + (1 - \alpha) \mu_2$ where $\alpha \in [0, 1]$ and experimental based constant that is adapted to find the highest number of subtle higher-order mutants.

Abuljadayel and Wedyan [**35**] measured the subtlety of a mutant *m* using the ratio between the number of test cases that kill *m* and the total number of test cases in the test suite. This metric range is between 0 and 1. According to this metric the subtle mutants are located close to 0 and the easy killed mutants exist close to 1.

6. CONCLUSION AND FUTURE WORK

In this paper, an empirical study to evaluate the impact of data-flow analysis on the subtlety of the higher-order mutants was introduced. The empirical study compared the data-flow based second order mutants and their constitute FOMs regarding the subtlety of each of them. In addition, it compared the data-flow based second order mutants and all du-pairs criterion regarding the subtlety of each of them. Therefore, it studies two research questions: which mutants are more subtle DFSOMs or their constitute FOMs mutants? And which mutants are harder to be killed or covered DFSOMs or all du-pairs criterion? The results of the conducted experiments showed that the subtlety of DFSOMs is higher than their FOMs by 6% in average. In addition, DFSOM criterion dominates all du-pairs criterion and covering DFSOM criterion is harder than covering all du-pairs criterion by 14.6% in average. The future work will focus on comparing the quality of test suite which covers all du-pairs against the quality of test suite which covers DFSOM.

ACKNOWLEDGMENT

This study was funded by the Deanship of Scientific Research, Taif University, KSA [1-438-5741].

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, 1978.

[2] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering SE-3*, vol. 4, pp. 279–290, 1977.

[3] W. B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *Journal of Systems and*

Software, vol. 83, pp. 2416–2430, 2010.

[4] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," *Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO '07, ACM*, pp. 1074–1081, 2007.

[5] A.T. Acree, "On Mutation," PhD thesis, Georgia Inst. of Technology, 1980.

[6] T. A. Budd, "Mutation Analysis of Program Test Data," PhD thesis, Yale Univ, 1980.

[7] A. J. Offutt and K. N. King, "A Fortran Language System for Mutation-Based Software Testing," *Software Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[8] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Journal of Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[9] Q. V. Nguyen and L. Madeyski, "Problems of Mutation Testing and Higher Order Mutation Testing," *Advances in Intelligent Systems and Computing*, vol. 282, pp. 157-172, 2014.

[10] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," in *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 249-258.

[11] E. Spafford et al., "Design of Mutant Operators for the C Programming Language," Purdue Univ., Technical Report SERC-TR-41-P 1989.

[12] A. P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," *In Proc. Fifth Int'l Computer Software and Applications Conf.*, pp. 604–605, 1991.

[13] G. Rotherme, A. J. Offutt, and C. Zapf, "An Experimental Evaluation of Selective Mutation.," *In Proc. 15th Int'l Conf. Software Eng.*, pp. 100–107, 1993.

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

[15] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Reducing the Cost of Higher-Order Mutation Testing," *Arabian Journal for Science and Engineering*, pp. 1-14, 2018.

[16] 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, 2014.

[17] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering Methodology*, vol. 1, pp. 5–20, 1992.

[18] M. Malevris, M. Kintis, and N. Papadakis, "Evaluating mutation testing alternatives: A collateral experiment," *In Proc. 17th Asia Pacific Soft. Eng. Conf. (APSEC)*, 2010.

[19] M. Garcia-Rodriguez, M. Polo, and I. Piattini, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Software Testing, Verification, and Reliability*, vol. 19, no. 2, pp. 111-131, 2009.

[20] A. S. Ghiduk, "Using Evolutionary Algorithms for Higher-Order Mutation Testing," *International Journal of Computer Science*, vol. 11, no. 2, pp. 93-104, 2014.

[21] A. S. Ghiduk, "Reducing the Number of Higher-Order Mutants with the Aid of Data Flow," *e-Informatica Software Engineering Journal*, vol. 10, pp. 31-49, 2016.

[22] M. Harman, Y. Jia, and W. B. Langdon, "A Manifesto for Higher Order Mutation Testing," *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pp. 80–89, 2010.

[23] M. Harman, Y. Jia, P. R. Mateo, and M. Polo, "Angels and monsters: an empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation," in *29th ACM/IEEE international conference on automated software engineering (ASE'14)*, 2014, pp. 397-408.

[24] E. Omar and S. Ghosh, "An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, 2012, pp. 1-9.

[25] E. Omar, S. Ghosh, and D. Whitley, "Constructing Subtle Higher Order Mutants for Java and AspectJ Programs," in *2013*

IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 2013, pp. 340 - 349.

[26] E. Omar, S. Ghosh, and D. Whitley, "Comparing Search Techniques for Finding Subtle Higher Order Mutants," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO '14)*, 2014, pp. 1271-1278.

[27] E. Omar, S. Ghosh, and D. Whitley, "Subtle Higher Order Mutants," *Information and Software Technology*, vol. 81, pp. 3-18, 2017.

[28] Q. V. Nguyen and L. Madeyski, "Problems of Mutation Testing and Higher Order Mutation Testing," *Advanced Computational Methods for Knowledge Engineering*, vol. 282, pp. 157-172, 2014.

[29] Q. V. Nguyen and L. Madeyski, "Searching for Strongly Subsuming Higher Order Mutants by Applying Multi-objective Optimization Algorithm," *Advanced Computational Methods for Knowledge Engineering*, vol. 358, pp. 391–402, 2015.

[30] Q. V. Nguyen and L. Madeyski, "Empirical Evaluation of Multi-Objective Optimization Algorithms Searching for Higher Order Mutants," *Cybernetics and Systems*, vol. 47, pp. 48-68, 2016.

[31] Q. V. Nguyen and L. Madeyski, "On the Relationship Between the Order of Mutation Testing and the Properties of Generated Higher Order Mutants," in *Asian Conference on Intelligent Information and Database Systems*, 2016, pp. 245-254.

[32] Q. V. Nguyen and L. Madeyski, "Higher Order Mutation Testing to Drive Development of New Test Cases: An Empirical Comparison of Three Strategies," in *Asian Conference on Intelligent Information and Database Systems*, 2016, pp. 235-244.

[33] Q. V. Nguyen and L. Madeyski, "Addressing Mutation Testing Problems by Applying Multi-Objective Optimization Algorithms and Higher Order Mutation," *Journal of Intelligent & Fuzzy Systems*, vol. 32, pp. 1173–1182, 2017.

[34] Q. V. Nguyen, "Is Higher Order Mutant Harder to Kill Than First Order Mutant? An Experimental Study," in *Asian Conference on Intelligent Information and Database Systems*, 2018, pp. 664-673.

[35] A. Abuljadayel and F. Wedyan, "An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms," *International Journal of Intelligent Systems and Applications*, vol. 1, pp. 34-45, 2018.

[36] M. Kintis and N. Malevris, "Using Data Flow Patterns for Equivalent Mutant Detection," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2014, pp. 196-205.

[37] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Higher order mutation testing: A Systematic Literature Review," *Computer Science Review*, vol. 25, pp. 29-48, 2017.

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

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

[40] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, pp. 367–375, 1985.

[41] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage," *14th Asia-Pacific Software Engineering Conference*, 2007.

[42] I. Burnstein, *Practical Software Testing: A Process-Oriented Approach. Springer.*, 2003.

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

[44] M. Papadakis and N Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Journal of Software Quality*, vol. 19, pp. 691–723, 2011.

[45] M. S. Hecht, *Flow Analysis of Computer Programs.*: Elsevier North Holland, New York, 1977.

[46] S. Rapps and E. J. Weyuker, "Data Flow

Analysis Techniques for Test Data Selection," in *6th international conference of software engineering*, 1982, pp. 272-278.

[47] F. E. Cocke and J. Allen, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, pp. 137-147, 1976.

[48] R. Malhotra, *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*.: Chapman & Hall/CRC, Taylor and Francis, 2015.

[49] B. A. Kitchenham et al., "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721-734, 2002.

[50] P. May, J. Timmis, and K. Mander, "Immune and Evolutionary Approaches to Software Mutation Testing," *LNCS 4628*, pp. 336–347, 2007.

[51] C. C. Michael, G. E. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software*, vol. 27, no. no.12, pp. 1085-1110, 2001.

[52] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test data generation using genetic algorithms," *Journal of Software Testing, Verifications, and Reliability*, vol. 9, pp. 263-282, 1999.

[53] Y. Ma and J. Offutt, "Description of Method-level Mutation Operators for Java," *https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf*, 2011.

[54] A. S. Ghiduk, *Software Test Data Generation Using Genetic Algorithms*, 1st ed.: LAP Lambert Academic Publishing, 2012.

[55] A. S. Ghiduk, "Automatic Generation of Object-Oriented Tests with a Multistage-Based Genetic Algorithm," *Journal of Computers*, vol. 5, no. 10, pp. 1560-1569, 2010.

[56] A. S. Ghiduk and M. R. Girgis, "Using Genetic Algorithms and Dominance Concepts for Generating Reduced Test Data," *Informatica Journal*, vol. 34, no. 3, pp. 377-385, 2010.

Table 8: The application of the guidelines for our empirical study.

Empirical area	Applicable guidelines
Experimental context	C1: In industrial case, describe entities, attributes, and measures for gathering contextual information.
	C2: Introduce the tested hypothesis and its theoretical background.
	C3: Describe the investigated questions and how these questions are addressed.
	C4: Describe the similar researches and how the current work relates to those researches.
Experimental design	D1: Define the population from which the subjects and objects are selected.
	D2: Describe the process used for selecting the subjects and objects.
	D3: Explain the process applied to assign the subjects and objects for treatments.
	D4: Control the design of the study to be close to designs analyzed in the statistical literature.
	D5: Describe the experimental unit.
	D6: Calculate the size of the required sample by carrying out a pre-experiment or pre-calculation.
	D7: Apply a proper level of blinding.
	D8: Avoid the self-evaluation of your work. If not, report what have implemented to minimize bias.
	D9: Use controls only when the control situation can be clearly well-defined.
	D10: Completely describe all treatments or actions and interventions.
	D11: Justify the use of specific metrics to measure the outcomes by showing the relevance between these metrics and the objectives of the empirical study.
Conduct of the experiment and data collection	DC1: For software, describe fully their all measures such as the entity, attribute, unit and counting rules.
	DC2: In subjective measures, describe the approaches applied to verify that the measurement is correct and consistent.
	DC3: Explain any quality control procedure which is used to prove the accuracy and completeness of data collection.
	DC4: In surveys, observe and report the rate of responses and explore their representativeness and the impact of non-responses.
	DC5: In observational empirical study, report the subjects that are dropped out from the study.
	DC6: In observational empirical study, retain data on the measures of performance which could be affected by the used treatment, even if they aren't the central issue of the study.
Analysis	A1: Identify clearly and definitely any procedure that is used to control the multiple testing.
	A2: Use blind analysis.
	A3: Carry out sensitivity analyses.
	A4: Verify that the data don't violate the hypotheses of the tests which are used on these data.
	A5: Verify the results by applying a proper quality control procedure.
Presentation of results	P1: Cite or give the details of all the statistical procedures which are used in the study.
	P2: Mention the statistical package which is used throughout the study.
	P3: Report the quantitative results and the significance levels.
	P4: If it is possible, present any raw data or confirm its availability to check by the reviewers.
	P5: Supply the reader by a proper descriptive statistics.
	P6: Use the graphics in an appropriate manner.
Interpretation of results	I1: Describe the population to which the inferential statistics and the predictive models are applied.
	I2: Distinguish the statistical significance against the practical importance.
	I3: Describe the type or category of the study.
	I4: Identify the limitations or drawbacks of the study.

Table 9: Method-level operators

Category	Operators B: Binary, U: Unary, S: Short-Cut	Possible Operation R: Replacement, I: Insertion, D: Deletion	Permutations	Total
Arithmetic	B (+, -, *, /, %)	R	5×4=20	
	U (+, -)	R, I, D	2×1+2×1+2×1=6	46
	S (op++, ++op, op --, -- op)	R, I, D	4×3+4×1+4×1=20	
Relational	B (>, >=, <, <=, ==, ! =)	R	6×5=30	30
Conditional	B (&&, ||, &, |, ^)	R	5×4=20	22
	U(!)	I, D	1×2=2	
Logical	B (&, |, ^)	R	3×2=6	8
	U(~)	I, D	1×2=2	
Assignment	S(+=, -=, *=, /=, %=, &=, |=, ^=)	R	8×7=56	56
Total of Possibilities				162

Table 10: No. of second-order mutants for each subject and their constituent.

Subject program	No. mutation at *def*	No. mutation at c-use	No. mutation at p-use	All permutations		Not selected yet	
				Second-order mutants	Constituent without duplication	Second-order mutants	Constituent without duplication
SP#1	65	50	46	2020	131	260	115
SP#2	120	60	50	660	230	110	70
SP#3	73	35	15	580	93	95	65
SP#4	100	93	31	2539	224	260	161
SP#5	118	104	29	2666	180	195	122
SP#6	91	91	25	2250	131	216	126
SP#7	85	121	25	2550	231	179	112
Total	652	554	221	13265	1220	1315	771

Table 11: Java example program and its data flow analysis.

Java function *Midnum*	Data flow actions	*dua* (variable, def, use)
0. public void Midnum(num1, num2, num3) {	-	# dua
1. **int** x, y, z;	1. -	1. (x, 3, 8)
2. **int** mid;	2. -	2. (x, 3,12)
3. x = num1;	3. x:def ; num1: c-use	3. (x, 3, 13)
4. y = num2;	4. y:def ; num2: c-use	4. (x, 3, 18)
5. z = num3;	5. z:def ; num3: c-use	5. (x, 3, 22)
6. mid = z;	6. mid:def ; z: c-use	6. (x, 3, 23)
7. **if**(y < z) {	7. y: p-use; z: p-use	
8. **if**(x < y) {	8. x: p-use; y: p-use	7. (y, 4, 7)
9. mid = y;	9. mid: def; y: c-use	8. (y, 4, 8)
10. }	10. -	9. (y, 4, 9)
11. **else** {	11. -	10. (y, 4, 18)
12. **if**(x < z) {	12. x: p-use; z: p-use	11. (y, 4, 19)
13. mid = x;	13. mid: def; x: c-use	
14. }	14. -	12. (z, 5, 6)
15. }	15. -	13. (z, 5, 7)
16. }	16. -	14. (z, 5, 12)
17. **else** {	17. –	15. (z, 5, 22)
18. **if**(x > = y) {	18. x: p-use; y: p-use	
19. mid = y;	19. mid: def; y: c-use	16. (mid, 6, 9)
20. }	20. –	17. (mid, 6, 13)
21. **else** {	21.-	18. (mid, 6, 19)
22. **if**(x > z) {	22. x: p-use; z: p-use	19. (mid, 6, 23)
23. mid = x;	23. mid: def; x: c-use	20. (mid, 6, 27)
24. }	24. –	
25. }	25.-	
26. }	26.-	
27. System.out.println(mid);	27. mid: c-use	Total = 20 *dua*
28. }	28.-	

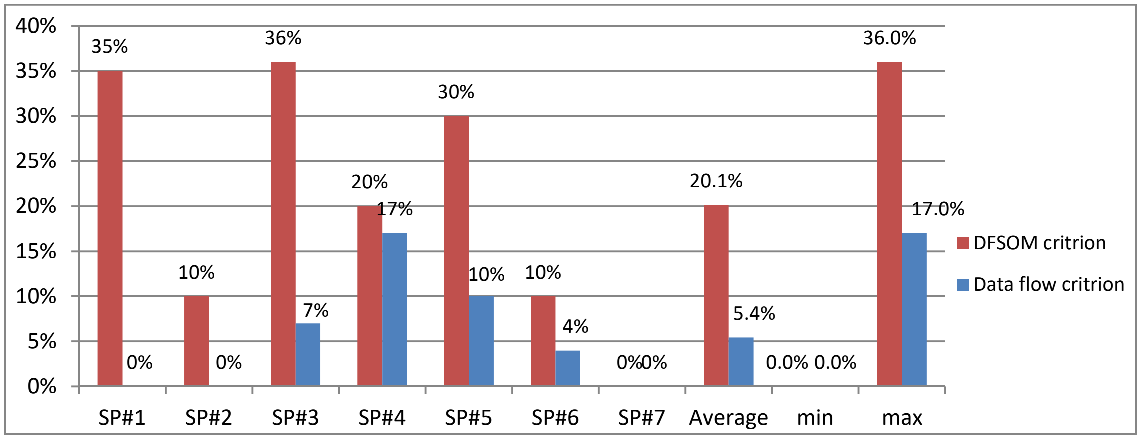

Figure 2: : Subtlety of DFSOM and Data flow criteria against the same test suite.

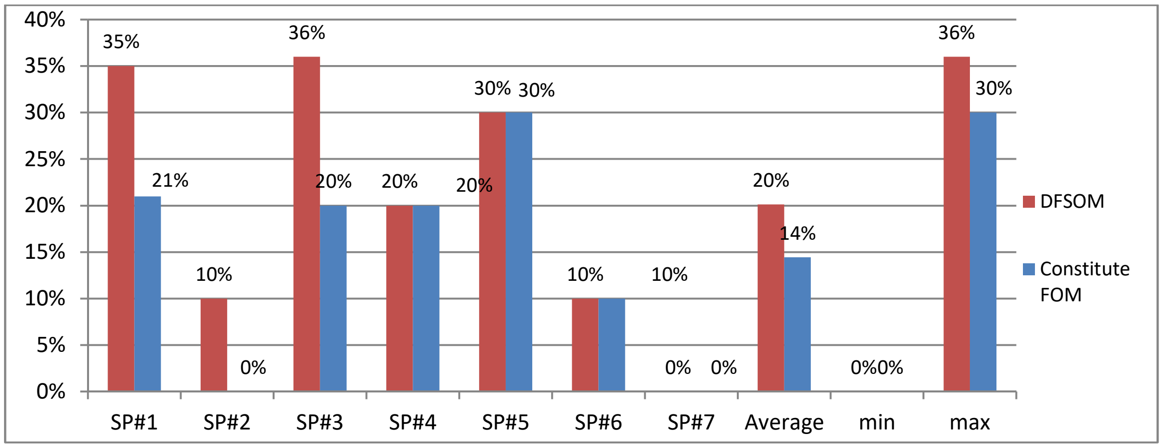

Figure 3: Subtlety of DFSOM and their Constitute FOM against the same test suite.