2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Using Mutants to Locate "Unknown" Faults

Mike Papadakis*
Interdisciplinary Center for Security, Reliability and Trust (SnT),
University of Luxembourg
Michail.Papadakis@uni.lu

Yves Le Traon
Interdisciplinary Center for Security, Reliability and Trust (SnT),
University of Luxembourg
Yves.LeTraon@uni.lu

***Abstract*—Many fault localization techniques operate by crosscutting coverage information of passed and failed test executions. Generally, their accuracy depends on the utilized coverage elements and on the selected test cases. This paper proposes a novel fault localization method using mutation and investigates its accuracy when using classical test selection criteria such as mutation, branch and block. A controlled experiment shows that (1) the mutation based approach is quite effective at identifying "unknown" faulty program statements. Additionally, the experimental results reveal (2) that the mutation-based test suites are significantly more effective at supporting fault localization than block or branch-based test suites. Further, (3) evidence in support of facilitating mutation alternatives, such as mutant sampling, in order to diminish mutation overheads is also given.**

Keywords- debugging, mutation analysis, fault localization

I. INTRODUCTION

Detecting, localizing and fixing bugs are essential software development activities. While software testing forms the main activity for detecting program defects, software debugging is the process of locating (diagnosing) and fixing the defective program parts. The fault localization process refers to the problem of identifying the defective program parts given the test execution failures. It has been recognized as one of the costlier parts of the debugging process which justify the important research effort for automating the fault localization.

When considering testing and fault detection, more than two decades of experiments on mutation testing have demonstrated that detecting artificial faults (e.g. seeded using mutation operators) allows effective detection of unknown, real ones, compared to more classical test selection criteria (e.g. based on code coverage). Test cases generated using mutations are good candidates for finding real faults.

When looking at diagnosis, mutants as relevant substitutes of real faults could be useful to improve fault localization activity. This raises the research questions of whether mutants could provide sufficient guidance for localizing "unknown" faults and whether test cases able to kill mutants could enable accurate fault localization.

Generally, fault localization approaches assist the programmers by giving some advice either on the causes of the failures or on the program locations that are responsible for some program failures. Some approaches i.e. Dynamic Slicing [1] produces a set of program statements that affect the failing program execution. Delta Debugging [2], [3], [4] tries to isolate the causes of program failures by examining the state differences between passing and failing program executions. Other techniques, usually referred to as coverage-based, [5], [6] monitor the program execution to gain runtime information, based on which they specify a suspiciousness rank of the program statements. Researchers have used many coverage elements such as the program statements [7], [8], [9] program branches [10], [6] program definition use pairs [11] and possible combinations of them [6], [12]. Empirical evidence has shown that coverage-based fault localization approaches can be very effective and helpful [9], [13] in diminishing the debugging effort.

Among the various coverage elements utilized by the fault diagnosis techniques, the most commonly used ones are the program statements and branches. Still, the use of mutants in locating program faults has drawn little attention by the researchers. This might attribute to the general belief that mutation testing is quite expensive and can not scale [14]. However, recent advances has shown that mutation testing can be practical [14], [15] and can be applied on real world applications [14], [15], [16]. Many efficient and scalable mutation testing tools such as the MiLu [17] and Javalanche [18] have been built with promising results. Further, by integrating the mutation analysis both in the testing and fault localization activities may keep the fault diagnosis expenses at a low level.

Mutation analysis works by introducing faults named mutants to the program under analysis. Mutants are produced based on simple syntactic rules, called mutation operators. Mutation testing is performed by executing the mutant programs with a selected set of test cases and by examining the differences in behavior between the mutant and the original program versions. Thus, the mutants can be categorized as killed and live. Killed mutants designate those that result in different with the original program version, outputs; while live are those of the opposite case. Mutation analysis relies on the assumption that mutants form "realistic" faults, even if artificially seeded. Several empirical results, such as Andrews et al. ones [19], provide evidence that this assumption is reasonable. Therefore, the following question can be positioned - if revealing mutants' results in revealing "unknown" faults, is the location of mutants able to assist the localization of "unknown" faults?

* This work was done while the author was at the Athens University of Economics and Business, Athens, Greece.

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.159

The present paper investigates this question, and eventually suggests the use of mutation analysis for fault localization. By utilizing mutants as alternatives to the structural code coverage, a novel mutation-based fault diagnosis approach can be defined. If validated, this approach may be used to kill two birds with one stone, meaning that mutation analysis could be reconciliate testing and diagnosis activities, which are usually targeting different objectives (fault detection and fault localization [20]). Minimizing the effort of the testing process requires the minimization/prioritization of the test cases while, minimizing the fault localization effort requires the maximization of the information provided by test execution.

This work aims to investigate a) whether mutation analysis can improve the effectiveness of coverage-based fault localization techniques, b) whether the use of mutation testing adequacy criterion can provide a sufficient and suitable set of test cases to effectively support the fault localization activity and c) to determine whether mutant sampling can be assisted for fault localization purposes. The above questions were explored on the Siemens benchmark program suite using its accompanied faulty versions.

The remainder of this paper is organized as follows: Section II and III present the underlying concepts and details the proposed approach. Its evaluation along with empirical results is described in Sections IV and V respectively. Sections VI and VII discuss about the proposed technique and its relation to the literature. Finally, the Section VIII concludes the paper and reports some future directions.

II. MUTATION TESTING AND FAULTY STATEMENTS

Provoking program failures forms the primary aim of the testing process. Developers when experiencing such failures move to the debugging phase that involves two main steps. The first one is to identify the faulty program places (diagnosis) and the second one is to fix those places. Adequacy or coverage criteria are usually utilized by testers in order to assist them with the testing process. Fault diagnosis techniques prioritize the program places in order to help testers locating faults. This Section summarizes the above concepts and techniques which underlies the work presented in this paper.

A. Code coverage and Mutation Analysis

Software testing process is performed by using a set of test cases based on which the software's behavior is exercised. Test adequacy criteria (also called coverage criteria) are employed in order to help testers select a small but representative of the whole possible cases, set of tests. This is approximated by possessing the requirement on the selected test cases to cover some specific program elements. Requirements on different elements form different criteria. The present paper considers block and decision criteria, that require from the test cases to cover-execute all program blocks and decisions. Test adequacy, here referred to as score or coverage level, is measured based on the ratio of the exercised, by the test cases, elements to the total ones.

Mutation analysis is a fault based technique. It is based on the hypothesis of the "competent programmer" i.e. the assumption that programmers produce programs that are nearly "correct" [21] and the "coupling effect" [21]. The coupling effect states that "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors". This assumption underlies the approach of the present paper in order to locate real-complex faults. By generating a mutant program two versions of the same program exists. The original one, say O and the mutated one, say M. If M is produced by making only one syntactic change to O it is called first order mutant. Otherwise it is called a higher order mutant [14]. This paper considers only first order mutants. Mutation can be used as a test adequacy criterion. This is accomplished by assessing the ability of test cases, say t, to distinguish the mutated from the original program versions. This distinction is usually performed by comparing the programs outputs, such as $O(t) \neq M(t)$. It is common to have situations where such cases do not exist. In this case the mutant M is called equivalent. The killing mutants' ratio is called mutation score and measures the adequacy of the test cases with respect to mutation testing.

A usual criticism of mutation is about its cost. Since a vast number of mutants are to be generated and executed with test cases, huge computational resources are needed. To overcome this problem researchers have suggested using various alternatives such as the "mutant sampling" [22] and the "selective mutation" [23]. In the "mutant sampling" approach a small percentage of mutants is sampled and considered as being the whole mutant set. In the "selective mutation" only mutants produced by specific operators are being considered. Empirical evidence [23], [24] has shown that both of the above approaches are capable of constructing high quality test data. More details about mutation and its alternatives can be found in [15] and [22].

B. Coverage-based Fault Localization

Research on Fault localization suggests that this process can be performed by utilizing the execution traces of the employed test suites. These approaches referred to as coverage-based [6] record the executed-covered code elements of the passing and failing test cases. The main idea exploiting by such approaches is that code elements executed by failing test cases are responsible for the failure. Thus, for each of the employed program elements they compute a suspiciousness value that approximates the probability of being faulty based on the frequency they appear in the failing and passing program executions. The programmer is assisted to find a fault by inspecting these highlighted elements in a decreasing order.

One of the most popular coverage-based methods is Tarantula [8]. Tarantula technique uses program statements as coverage elements and computes their suspiciousness by using a formula similar to the one presented in Table I. It is noted that these values are within the range of [0-1]. Going along the same lines, other formulas can be defined. Abreu et al. [7] investigated this issue and concluded that a similarity coefficient named *Ochiai* was the most effective one. Santelices et al. [6] also report that their experiments supported the use of this formula and thus they used it in

their approach. The *Ochiai* suspiciousness calculation formula for a code element *e* is presented in Table I. It is noted that the same formula is employed by the proposed mutation-based approach.

To demonstrate how Tarantula works consider the example of Fig. 1, which has been taken from the work of Santelices et al. [6]. For the discussion, focus only on the upper part (above the black line) of the figure that corresponds to the statement-based approach. The lower part of the figure corresponds to the mutation based approach that is discussed in the next section. The example program (*mid*) has 13 statements (column Statements) and is executed with six tests (top of the columns Test 1-6). Test columns record the execution traces (denoted with "1" per executed statement) of the respective tests. The columns labeled as "#Passed" and "#Failed", record the number of passing and failing test cases (denoted as *passed*(*e*) and *failed*(*e*) in the *Ochiai* formula of Table I) that execute each program statement. The columns labeled as "Suspiciousness" and "Rank" record the suspiciousness scores (calculated by the *Ochiai* formula) and the respective ranking of each statement. Fault 1 (localized at statement 3) is detected by two test cases (bottom of the columns Test 1-6) and is ranked in the 6th position while Fault 2 (localized at statement 7) is detected by one and it is ranked in the 1st position.

As pointed out before, after the localization process the programmer has to check the ranked statements in a decreasing suspiciousness order in order to find and fix the fault. Hence, ranking faulty statements at a higher position in the ranked order results in putting less effort by the programmer to find the error.

III. RANKING STATEMENTS USING MUATION ANALYSIS

A. Using mutants to locate "unknown" faults

This section discusses the use of mutants to assist the fault localization process. We call an "unknown fault" a fault which has been detected by at least one test case, but that has still to be located. A mutant M1 is said to have a behavior similar to another mutant M2 if M1 and M2 are killed by (almost) the same test cases. The proposed approach is motivated by the following observations:

- Mutants located on the same program statements frequently exhibit a similar behavior**.**
- Mutants located in diverse program statements exhibit different behaviors. For a 'hard-to-kill' mutant, a test case that kills it is usually specific, and able to kill mutants located on the same statement.

Consider a scenario where the program under test contains an "unknown" fault that is in fact a mutant, not used by the mutation approach. Based on the above observations this fault is likely exercised similarly to other mutants applied at the same statement and differently to those located in other statements. Then, within this assumption the "unknown and seeded faults exhibit similar behaviors", the identification of an "unknown" fault may be obtained thanks to a mutant fault at the same (or close) location. Next section provides an example of the proposed approach illustrating the above scenario.

TABLE I. THE OCHIAI FORMULA

$Suspiciousness(e) = \frac{failed(e)}{\sqrt{totfailed * (failed(e) + passed(e))}}$
Where: *totfailed* - the total number of test cases that fail, *failed*(*e*) - the number of test cases that cover the code element *e* and fail and *passed*(*e*) - the number of test cases that cover the code element *e* and pass.

Thus, the intuition behind the proposed approach is this implicit link of the behavior of "unknown" faults with some mutants. Based on the location of the mutants, one can localize real "unknown" faults. A way to achieve a localization approach from that intuition can be to extend the coverage-based fault localization techniques, using mutants instead of structural code elements (such as statements or decisions). Thus, measuring the number of killed mutants by the passing and failing test executions one can get an indication about the suspiciousness of those mutants. This can be computed straightforwardly by applying the *Ochiai* formula (Table I) with coverage elements (*e*) some mutants. Killed mutants are treated as covered elements (*e*) while the live ones are ignored i.e. treated as uncovered elements.

The proposed approach considers only first order mutants and relies on the coupling effect in order to locate complex faults. The use of first order mutants helps assigning suspiciousness values on the program statements. Since mutants are produced based on simple syntactic rules each mutant is located at one statement. Thus, the suspiciousness values computed for the mutants can be assigned to their respective statements. However, most program statements involve many mutants and assigned suspiciousness values must be combined. In this paper, the maximum suspiciousness value of its respective mutants is assigned to this statement. Statements without mutants are assigned with the worst suspiciousness value (the number of program statements). This was done in order to indicate that these statements will be among the last ones to be inspected. The need for mapping those statements with suspiciousness values [6] is not crucial since mutants operate on most statements. Both the above issues, the assignment of suspiciousness values on program statements and the use of mappings between the various mutant elements [6], constitute a matter of further research. This paper only focuses on a first investigation of the intuition that artificial faults (mutants) can help locating unknown real ones.

B. An Ilustrative example

Consider the example of Fig. 1, and focus on the bottom part (below the black line) of the figure, that illustrates the use of mutants in localizing faults. This example shows:

1. how the proposed approach works,
2. a concrete scenario of mutant-fault localization using different types of mutants.

Fault 1 (localized in statement 3) is due to extra code fragments ($y < z \rightarrow y < z$ - *1*) and can be precisely localized using the relational mutant operator i.e. it changes the instance of relational operators with the other ones. The utilized mutant elements are demonstrated in the column Mutants and they are named as *M1-M35*. Fault 2 (localized in statement 7) is an assignment expression error (m = $x \rightarrow m$

= *y*) and can be localized using numerical constant increment and decrement mutants i.e. it add and subtracts a constant value to a program's variable. The mutant elements used are presented in the column Mutants and they are named as *M1-M32*. The Test columns record the killed mutants (denoted with "1" per killed mutant) by the test cases. The columns labeled as "#Passed" and "#Failed" record the number of passing and failing test cases (denoted as *passed*(*e*) and *failed*(*e*) in the *Ochiai* formula of Table I) that kill each considered mutant. The columns labeled as "Suspiciousness" and "Rank" record the suspiciousness scores (calculated by the *Ochiai* formula) and the respective statement rankings.

Fault 1 (localized at statement 3) is detected by two test cases (bottom of the columns Test 1-6) and is ranked in the 1th position (Rank column). The *M1* mutant was only killed by the two failing test cases, having a suspiciousness value *1.0*. Fault 2 (localized at statement 7) is detected by one test and it is ranked in the 1st position based on the suspiciousness values (*0.71*) of the *M17* and *M18* mutants.

Conclusively, Fig. 1 demonstrated how two faults can be effectively localized based on mutation analysis.

IV. Experimental Study

This section describes the empirical setup and evaluation of the proposed approach. First, it describes the definition of the conducted experiment by setting out the research questions under investigation. Then, details about the selected subjects and tools are given. Finally, a description of the experimental setup and analysis is provided.

A. Definition of the Experiment

The present study seeks to empirically investigate the following research questions (RQs):

- **RQ1:** *How effective is the mutation-based fault localization approach? Is this approach more effective in assisting fault localization process than the statement-based one?*
- **RQ2:** *What is the impact of test adequacy criteria on the effectiveness of mutation and statement based fault localization techniques? Comment, in this study Block, Branch and mutation adequate test suites were used.*
- **RQ3:** *How is the effectiveness of mutation-based fault localization technique affected by using different mutant sets? Comment, in this study random sampling of 10%, 20%, 30%, 40% and 50% mutant sets were used.*

Taking into account RQ1 and showing that the effectiveness of fault localization techniques can be improved will benefit researchers in seeking ways to reduce the program debugging expenses. Answering RQ2 is important in order to show whether the use of testing adequacy criteria is practical for the fault localization process. This answer will indicate whether programmers should put effort on localizing faults directly after the testing process or if they should produce some additional tests first. Additionally, RQ2 will give an answer whether mutation based localization approach is worthwhile when employing a basic testing approach such as block or branch coverage. RQ3 forms one fist step towards dealing with mutation analysis expenses. If only a small loss on localization effectiveness is observed when few mutants are considered, it offers a practical answer to the computational cost that full mutation analysis usually requires.

B. Subject Programs and test suite pools

The conducted experiment employed the benchmark programs of the Siemens suite which have been widely used in mutation testing and fault localization experiments e.g. [2], [6], [9], [10], [12], [25]. The suite is composed of seven programs written in C and is accompanied by test suite pools and a set of 132 faults. One fault was excluded from the considered set, since it did not result in any execution failure, mandatory requirement of the examined localization methods. This action was also taken on other similar studies e.g. [6], [9], [25]. Table II records details about the program lines of code (LOC), the size of the test pools and the number of faults per program. These programs were chosen due to their widespread use in the literature on the one hand and their availability along with the accompanied test and fault sets from the Software-artifact Infrastructure Repository at the University of Nebraska-Lincoln [26] on the other.

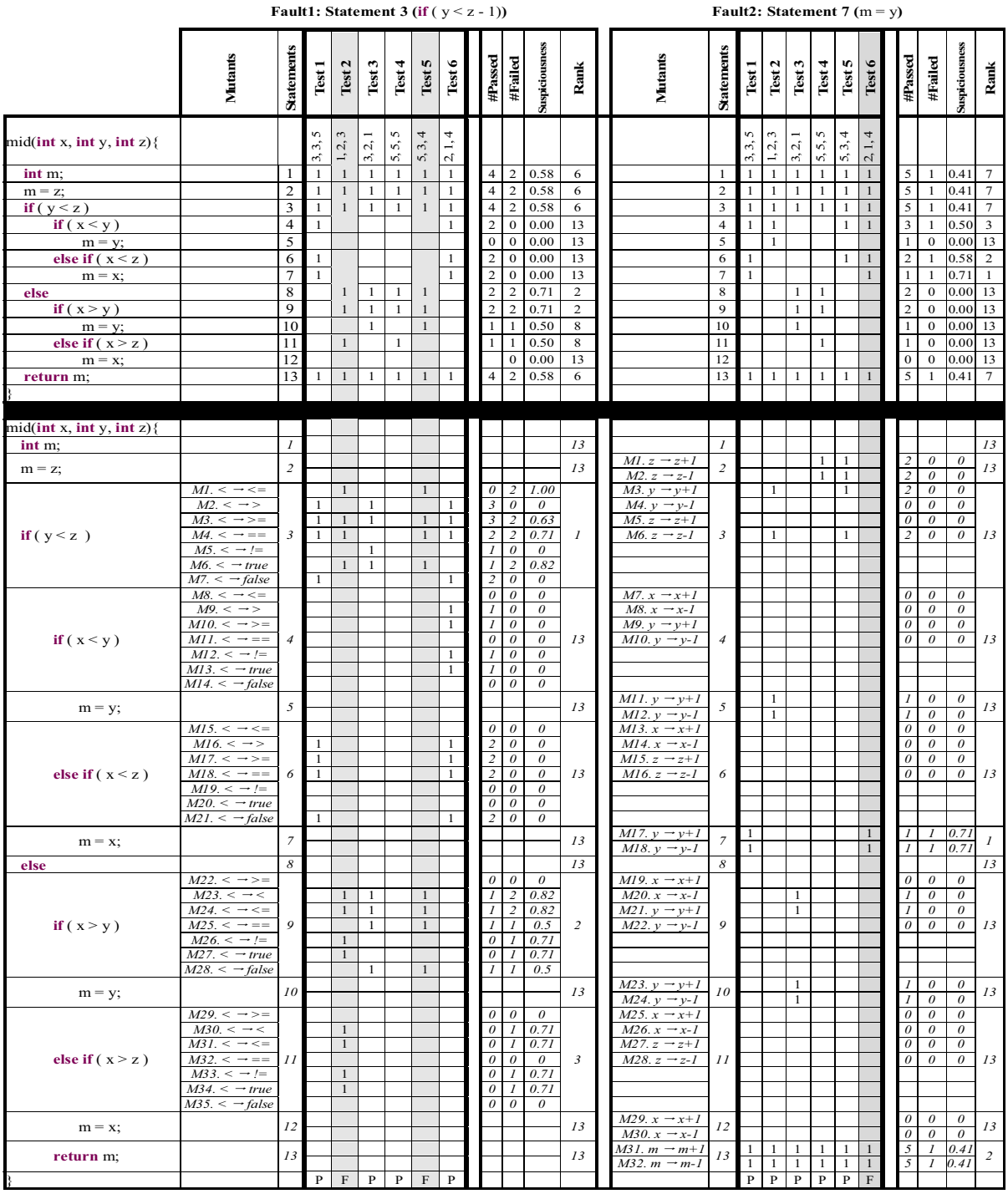
The program suite was initially employed in an empirical study by Hutchins et al. [27] for comparing various structural testing criteria. Later, it was extended and adapted appropriately from other researchers to support their experiments [28]. According to Hutchins et al. [27] the accompanied set of faults was manually produced by various researchers with the intention of introducing realistic faults. The accompanied test suites were produced based on a combination of both black and white box approaches such as random, category-partition, statements, decisions and definition-use pairs, with the aim of producing a comprehensive and suitable for empirical studies test suite. More details about the construction of the test suite pools can be found in Harder et al. [28].

C. Utilized tools and implemetation details

The present study used the Proteum [1] mutation testing system, by Maldonado et al. [29] in order to support the mutation analysis process. To gather the required tracing information a prototype has been implemented on top of the Wet [30] framework in the same lines utilized in [12]. Wet works at machine code instructions' granularity and thus, it collects the required information in terms of instruction instances. The Instruction terms are mapped to their respective program statements which are identified based on their line numbers. The prototype implements both the statement-based and mutation-based approaches utilizing the *Ochiai* formula (given in Table I).

The ATAC [31] coverage tool was used for the selection of the Block and Branch test sets from the accompanied test pools and Proteum [29] for the mutation ones. These tools have also been used in software testing experiments e.g. [19], [15], [24]. Details about the test selection process are given in the following subsection.

[1] Proteum/IM 2.0 was used by utilizing only the unit level operators.



	Mutants	Statements	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	#Passed	#Failed	Suspiciousness	Rank
Fault1: Statement 3 (if (y < z - 1))												
mid(int x, int y, int z){			3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	2, 1, 4				
int m;		1	1	1	1	1	1	1	4	2	0.58	6
m = z;		2	1	1	1	1	1	1	4	2	0.58	6
if (y < z)		3	1	1	1	1	1	1	4	2	0.58	6
if (x < y)		4	1					1	2	0	0.00	13
m = y;		5							0	0	0.00	13
else if (x < z)		6	1					1	2	0	0.00	13
m = x;		7	1					1	2	0	0.00	13
else		8		1	1	1	1		2	2	0.71	2
if (x > y)		9		1	1	1	1		2	2	0.71	2
m = y;		10			1		1		1	1	0.50	8
else if (x > z)		11		1		1			1	1	0.50	8
m = x;		12								0	0.00	13
return m;		13	1	1	1	1	1	1	4	2	0.58	6
}												
Fault2: Statement 7 (m = y)												
mid(int x, int y, int z){			3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	2, 1, 4				
int m;		1	1	1	1	1	1	1	5	1	0.41	7
m = z;		2	1	1	1	1	1	1	5	1	0.41	7
if (y < z)		3	1	1	1	1	1	1	5	1	0.41	7
if (x < y)		4	1				1	1	3	1	0.50	3
m = y;		5		1					1	0	0.00	13
else if (x < z)		6	1				1	1	2	1	0.58	2
m = x;		7	1					1	1	1	0.71	1
else		8			1	1			2	0	0.00	13
if (x > y)		9			1	1			2	0	0.00	13
m = y;		10			1				1	0	0.00	13
else if (x > z)		11							1	0	0.00	13
m = x;		12							0	0	0.00	13
return m;		13	1	1	1	1	1	1	5	1	0.41	7
}			P	F	P	P	F	P				
mid(int x, int y, int z){												
int m;		1										13
m = z;		2										13
if (y < z)	M1. < → <=	3		1			1		0	2	1.00	1
	M2. < → >		1		1			1	3	0	0	
	M3. < → >=		1	1	1		1	1	3	2	0.63	
	M4. < → ==		1	1			1	1	2	2	0.71	
	M5. < → !=				1				1	0	0	
	M6. < → true			1	1		1		1	2	0.82	
	M7. < → false		1					1	2	0	0	
if (x < y)	M8. < → <=	4							0	0	0	13
	M9. < → >							1	1	0	0	
	M10. < → >=							1	1	0	0	
	M11. < → ==								0	0	0	
	M12. < → !=							1	1	0	0	
	M13. < → true							1	1	0	0	
	M14. < → false								0	0	0	
m = y;		5										13
else if (x < z)	M15. < → <=	6							0	0	0	13
	M16. < → >		1					1	2	0	0	
	M17. < → >=		1					1	2	0	0	
	M18. < → ==		1					1	2	0	0	
	M19. < → !=								0	0	0	
	M20. < → true								0	0	0	
	M21. < → false		1					1	2	0	0	
m = x;		7										13
else		8										13
if (x > y)	M22. < → >=	9							0	0	0	2
	M23. < → <			1	1		1		1	2	0.82	
	M24. < → <=			1	1		1		1	2	0.82	
	M25. < → ==				1		1		1	1	0.5	
	M26. < → !=			1					0	1	0.71	
	M27. < → true			1					0	1	0.71	
	M28. < → false				1		1		1	1	0.5	
m = y;		10										13
else if (x > z)	M29. < → >=	11							0	0	0	3
	M30. < → <			1					0	1	0.71	
	M31. < → <=			1					0	1	0.71	
	M32. < → ==								0	0	0	
	M33. < → !=			1					0	1	0.71	
	M34. < → true			1					0	1	0.71	
	M35. < → false								0	0	0	
m = x;		12										13
return m;		13										13
}			P	F	P	P	F	P				
mid(int x, int y, int z){												
int m;		1										13
m = z;		2										13
if (y < z)	M1. z → z+1	3					1	1	2	0	0	13
	M2. z → z-1						1	1	2	0	0	
	M3. y → y+1			1			1		2	0	0	
	M4. y → y-1								0	0	0	
	M5. z → z+1								0	0	0	
	M6. z → z-1			1			1		2	0	0	
if (x < y)	M7. x → x+1	4							0	0	0	13
	M8. x → x-1								0	0	0	
	M9. y → y+1								0	0	0	
	M10. y → y-1								0	0	0	
m = y;	M11. y → y+1	5		1					1	0	0	13
	M12. y → y-1			1					1	0	0	
else if (x < z)	M13. x → x+1	6							0	0	0	13
	M14. x → x-1								0	0	0	
	M15. z → z+1								0	0	0	
	M16. z → z-1								0	0	0	
m = x;	M17. y → y+1	7	1					1	1	1	0.71	1
	M18. y → y-1		1					1	1	1	0.71	
else		8										13
if (x > y)	M19. x → x+1	9							0	0	0	13
	M20. x → x-1				1				1	0	0	
	M21. y → y+1				1				1	0	0	
	M22. y → y-1								0	0	0	
m = y;	M23. y → y+1	10			1				1	0	0	13
	M24. y → y-1				1				1	0	0	
else if (x > z)	M25. x → x+1	11							0	0	0	13
	M26. x → x-1								0	0	0	
	M27. z → z+1								0	0	0	
	M28. z → z-1								0	0	0	
m = x;	M29. x → x+1	12							0	0	0	13
	M30. x → x-1								0	0	0	
return m;	M31. m → m+1	13	1	1	1	1	1	1	5	1	0.41	2
	M32. m → m-1		1	1	1	1	1	1	5	1	0.41	
}			P	P	P	P	P	F				

Figure 1. Fault localization example using program statements and mutants. The upper part coresponds to a statement-based approach while the bottom part coresponds to the muation-based one.

D. Experimental Regime

The following experiment was set to address the stated RQs.

Initially all subject programs (including the faulty ones) were executed with all the available test cases in order to record the passing and failing executions of the entire test suite. Then, execution traces of all available test cases per subject program were collected. These traces were used in order to produce the statement-based fault localization results (per utilized fault). The study of RQ1 and RQ3 was based on these results.

TABLE II. DESCRIPTION OF THE SELECTED SUBJECTS

Program Name	LOC	whole Test Suite	Number of Faults
Schedule	296	2650	9
Schedule2	263	2710	10
Tcas	137	1608	41
Totinfo	281	1052	23
Printtokens	343	4130	7
Printtokens2	355	4115	10
Replace	513	5542	32

With respect to the mutation-based approach (examined by RQ1), the whole set of utilized mutants were generated, compiled and executed against the entire provided test suite pool. This process determined the killed and live mutants per test case, information used by the proposed approach in order to compute mutant suspiciousness and produce mutation-based fault localization results. Mutant sampling approach (examined by RQ3) was performed by selecting and generating at random, only a percentage of the whole set of mutants. Five different sampling ratios were considered (10%, 20%, 30%, 40% and 50%). In order to avoid any bias from the sampling process, 10 independent sets of mutants, per utilized ratio, were sampled, resulting in 50 mutant sets in total. For each one of those 50 mutant sets the same process as with the whole set of mutants was followed.

One of the aims of this study, regarding RQ2, is to investigate the ability of the examined fault localization methods in localizing a detected fault when using adequate (with respect to testing criterion) test sets. Thus, the utilized test sets should expose the considered fault and being adequate[2] at the same time. This study considers block, branch and mutation testing criteria. In order to avoid any side effects through the random selection of test cases, 10 independent test sets were constructed. Thus, the experiment considers in total 1310 test sets (131 faults × 10 test sets) per utilized testing criterion. The test sets were constructed from the available test suite pool using the procedure of Fig. 2. The term score refers to the utilized criterion coverage, such as the block, or branch or the mutation score for the case of mutation. Additionally, 10 test sets per utilized fault were constructed based on random selection from the available test pool. These sets, denoted as Random, were of the same size with the mutation ones and used to determine whether they have similar effects on fault localization with the mutation ones.

Only the executable statements were ranked in the present experiment due to the function of the developed tool (see Section IV.*C*). Additionally, to complete it with reasonable resources (vast resources are typically needed by mutation analysis) the localization process was performed on the main program versions. Further, in this experiment, statements with the same suspiciousness value are ranked together at the upper of their ranks. For instance, statements 8 and 9 of the Fig. 1 have the highest suspiciousness value (0.71), for fault 1, but they are both assigned with a rank of 2 (instead of ranks 1 and 2). This is a typical approach in the literature in this kind of experiments e.g. [6], [9], [25].

Comparing the accuracy of the examined methods between the different programs, a score for diagnosis effectiveness should be adopted. The most commonly used score by the literature in such cases is the one proposed by Jones et al. [9] in evaluating the Tarantula fault localization system. The "*score*" measures the percentage of executed program statements that need not to be examined if statements are examined by the programmer in a decreasing suspiciousness order. The use of the "*score*" is based on the assumption that the programmer will inspect each program statement until finding the faulty one based on the order specified by the fault localization tool. Along these lines the "score" value is calculated based on the following formula:

$$"score" = \frac{total\ executed\ statements - rank}{total\ executed\ statements}$$

In the above formula, rank indicates the position of the faulty statement in the ranked list produced by the fault localization method. Greater "score" values suggest that less program code needs inspection by the programmer in order to identify the sought fault. Similarly, the term cost refers to the ratio of a given rank of a faulty statement to the total number of executed statements.

Lastly, some additional concerns were made about the utilized faults and their localization. In cases of faults that involve omitted statements, it was assumed that these faults are found if the programmer inspects a statement next to the missing statement. Otherwise, there will never be such a mutated or executed statement. Similar situation is experienced in the cases of faults occurring on non executable statements such as variable initializations or constant assignment statements. In such cases, the faulty statements will not result in the considered suspiciousness list (for both of the utilized approaches). Hence, it was assumed that these faults will be located whenever the programmer inspects a statement using the constant or the faulty defined variable.

```
Input: Test suite pool score PoolScore of the aimed criterion
Output: Adequate test set with respect to the aimed criterion
Set CurrSet = [ ];
SetScore = 0;
select one test case (TC) able to expose the considered fault and put
it in the CurrSet. The selection was performed at random among the
available tests that expose the considered fault;
while ( SetScore < PoolScore ){
    add to CurrSet a randomly selected test case (TC) from the pool
    Execute the CurrSet and determine its score (CurrScore) level
    if ( SetScore < CurrScore )
        SetScore = CurrScore;
    else
        remove TC from the CurrSet;
}
return the CurrSet;
```

Figure 2. Test selection Procedure

[2] A test set is considered to be adequate if it achieves the same level of coverage with the whole suite.

Figure 3. Effectiveness comparison of the mutation-based and statement-based fault localization methods using the whole test suite

V. EXPERIMENTAL RESULTS

This section reports results on performing statement and mutation based fault localization methods according to the process specified in the previous section.

A. *Effectiveness Evaluation – (RQ1)*

The effectiveness results of mutation and statement based approaches, with respect to RQ1, are summarized in the graph of Fig. 3. The obtained results are categorized based on the assigned "scores", practice also used in [12], [13] and [25], in the following categories: *99-100%, 90-100%, 80-100%, 70-100%, 60-100%, 50-100%, 40-100%, 30-100%, 20-100%, 10-100%, 0-100%*. Fig. 3 plots the percentage of faults effectively localized (*y – axis*) within the given range, "score" categories, the percentage of statements not needing inspection (*x – axis*) by the programmer. Thus, method curves (data points) that appear higher in the graph reflect a better fault localization effectiveness. For example a programmer is able to effectively localize approximately *0.90* of the total faults when using the mutation-based and only *0.44* with statement-based approaches, by inspecting only a *10%* of the programs' code[3]. Following these lines Table III in the columns "StLoc whole-suite" and "MutLoc whole-suite" records these results. Mutation-based approach achieved to effectively localize the *0.11, 0.89 and 0.93* of faults in the *99%, 90% 80%* categories while the statement based one *0.41, 0.44 and 0.60* respectively. These experiments indicate that the mutation-based approach outperforms the statement-based one. This difference is of practical significance: the average "score" (average "score" values of all the examined faults) of the statement-based approach is equal to *77%* while the mutation-based one *95%*.

Fig. 4 depicts the fault localization cost per fault. From this graph it can be observed that the cost for localizing faults with mutation-based approach is lower in most cases. Additionally, the difference is considerable for most cases. Further, in the sample of the 131 examined faults, mutation-based approach performed better in the 108 cases, worst in 17 cases and had equal effectiveness in 6 cases. An examination of these 17 cases reveals that in 9 cases this difference was less than 1%. Only in 4 cases the statement-based approach was better by more than 5% but no more than 11%. These results suggest that whenever the statement-based approach achieves a better effectiveness this difference is not so important.

B. *Testing criteria and fault localization – (RQ2)*

Fault localization approaches rely on coverage of program' elements and the utilized number of test cases. Consequently, it seems natural to expect that using testing criteria requiring more test cases (such as mutation testing) will assist the localization of faults. However, since the mutation-based fault localization method relies on the killed mutants, it is expected (intuitively) to observe a low effectiveness when many mutants are not killed by the employed tests. Thus, low quality test suites such as those coming from block and branch testing should greatly affect the effectiveness of the localization method.

The obtained results to address this issue, RQ2, are recorded in Table III and Fig. 5. Table III and Fig. 5 present the ratio of the effectively localized faults at the various considered "score" ranges when using the Block (Block-suite), Branch (Branch-suite), Mutation (Mut-suite) and Random (Rand-suite) test suites by employing both statement (StLoc) and mutation (MutLoc) based fault localization methods. Additionally, Table III records the obtained results for the whole suite (whole-suite). These results confirm the intuition that the use of 'more effective at revealing faults' testing criteria helps also the localization process. Both examined localization approaches experience considerably better effectiveness when utilizing test suites adequate with respect to mutation than those based on Random, branch or block criteria.

The most interesting finding of these results is that mutation-based localization approach out-performs the statement-based one in all cases, even when using block adequate test suites. Further, it was found that this difference is statistically significant [4] in all cases. Moreover, mutation-based approach is more effective with block suites than the statement-based one with the whole suite. Recall, that the whole suite is a relatively huge and comprehensive one, see section IV.*B* for details.

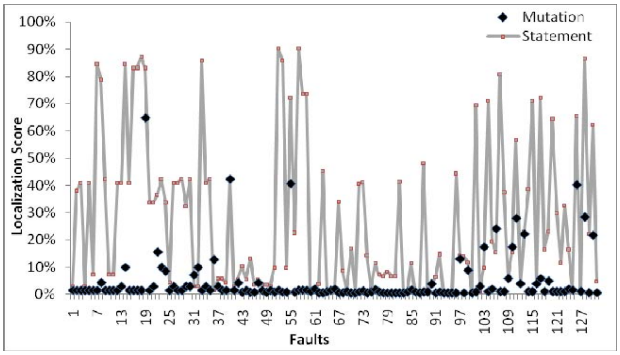


Figure 4. Fault localization cost per utilized fault

[3] The results consider only the executable statements not the whole program ones.

[4] Statistical paired t test with statistical significance difference: $p < 0.001$

TABLE III. AVERAGE FAULT LOCALIZATION SCORES FOR THE BLOCK, BRANCH, RANDOM, MUTATION AND THE WHOLE TEST SUITES.

Score	StLoc Block-suite	StLoc Branch-suite	StLoc Mut-suite	StLoc Rand-suite	StLoc whole-suite	MutLoc Block-suite	MutLoc Branch-suite	MutLoc Mut-suite	MutLoc Rand-suite	MutLoc whole-suite
99-100%	0.01	0.02	0.08	0.05	0.11	0.01	0.01	0.28	0.19	0.41
90-100%	0.23	0.32	0.44	0.35	0.44	0.49	0.56	0.88	0.82	0.89
80-100%	0.32	0.39	0.52	0.51	0.60	0.79	0.80	0.94	0.92	0.93
70-100%	0.44	0.50	0.60	0.56	0.66	0.88	0.91	0.96	0.94	0.97
60-100%	0.54	0.66	0.69	0.67	0.75	0.95	0.96	0.98	0.98	0.97
50-100%	0.69	0.79	0.81	0.79	0.85	0.98	0.98	0.98	0.98	0.99
40-100%	0.79	0.83	0.84	0.84	0.89	0.99	0.99	0.99	0.99	0.99
30-100%	0.85	0.85	0.85	0.87	0.90	1.00	1.00	1.00	1.00	1.00
20-100%	0.91	0.91	0.92	0.91	0.93	1.00	1.00	1.00	1.00	1.00
10-100%	0.99	0.99	1.00	0.98	1.00	1.00	1.00	1.00	1.00	1.00
0 -100%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Another interesting finding was the noticeable improvement on the fault localization approaches effectiveness, especially that of the mutation-based one, when mutation adequate test suites are employed. The question that it is raised here is whether this improvement is attributed to the size of the utilized test sets and not to their adequacy. To examine this issue, the average size of the selected test sets per considered criterion was computed and presented at Table IV. From this table it can be observed that Mutation tests are considerably more than those of Branch and Block. However, the results of Table III and Fig. 5, reveal that Mutation test sets have an advantage over the Random ones. Recall that Random test sets are of the same average size with the mutation ones. Further, this advantage is of statistical significance [5] suggesting that mutation test suites are suitable for assisting fault localization. Considering whether mutation adequate test suites can be improved to assist further the fault localization process a comparison between the mutation test sets and the whole test suite was performed. The results suggested that in the case of statement-based method the improvement was more evident. In the case of mutation-based method the whole suite provide a slight improvement on the method's effectiveness. In both cases the difference was not of great statistical significance. This finding suggests that there is a slight room for improvement in the methods' effectiveness by producing additional test cases.

C. Mutant sampling evaluation – (RQ3)

Mutation analysis has been identified as a costly technique. To overcome its difficulties, various mutation alternative techniques have been proposed [22], [15]. The present study examines the use of mutant sampling in fault localization. Fig. 6 presents the average effectiveness results of the mutant sampling technique with sampling ratios 10%, 20%, 30% 40% and 50%. For evaluation reasons Fig. 6 also plots the results of the whole utilized mutant set (denoted as 100%) and the statement-based ones. The graph of Fig. 6 suggests that all the examined sampling ratios experience loss in their effectiveness compared to the whole mutant set. In the case of 10% this loss is more apparent than the rest utilized approaches, which have a similar effectiveness.

[5] Statistical paired t test with statistical significance difference: $p < 0:001$

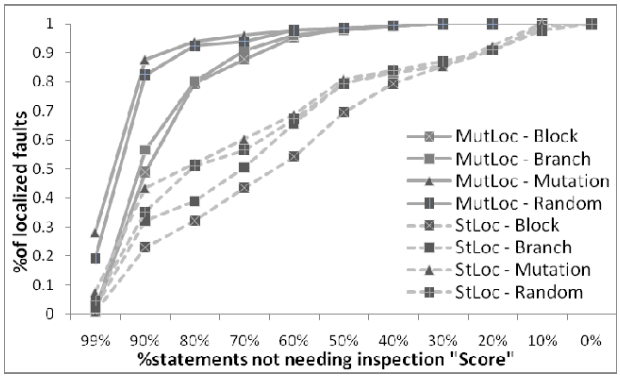


Figure 5. Effectiveness comparison of the mutation-based (MutLoc) and statement-based (StLoc) fault localization methods by utilizing Block, Branch and Mutation and Random test sets

By statistically comparing the differences between the various sampling ratios it was found that only the 10% and 20% sampling ratios have statistically significant differences with the whole set of mutants. However, the 10% mutant sampling approach outperforms the statement-based one with great statistical significance. On average 10% mutant sampling achieved to effectively localize 0.89 of the introduced faults while the statement-based one only the 0.77 of them. In view of this, it can argue that mutation alternative methods can be effectively utilized to assist the fault localization process.

VI. DISCUSSION

The main problem faced by researchers when employing mutation analysis is about its scalability. The question that it is raised here is whether the proposed approach can scale to larger real world programs. This issue is a matter of further investigation, i.e. the optimal subset selection of mutants capable to effectively locate

TABLE IV. AVERAGE TEST SUITE SIZE

Program Name	Block Tests	Branch Tests	Mutation Tests
Schedule	4.54	7.33	28.91
Schedule2	5.66	8.24	35.23
Tcas	5.46	9.34	74.65
Totinfo	6.34	6.66	29.88
Printtokens	9.57	10.54	33.09
Printtokens2	8.62	11.02	28.17
Replace	13.98	18.50	145.92

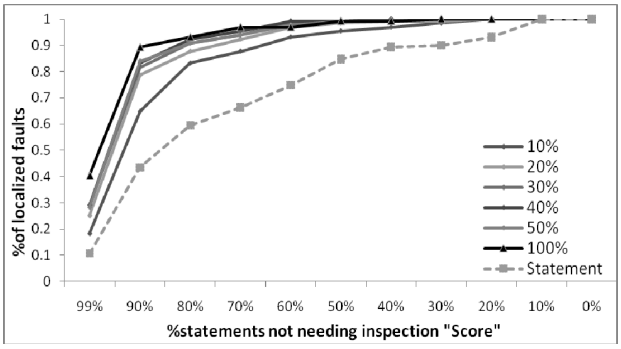


Figure 6. Mutat sampling approaches in assisting fault localization

program faults at a low computational cost. However, the obtained results suggest that mutant sampling can be used as an alternative mutation method, in locating program faults. Other, possible direction in dealing with the scalability of the proposed approach is due to its combined use of other methods such as dynamic slicing [1].

Generally, fault localization is performed after the testing process. Conversely, if mutation was employed during the testing phase, then most and perhaps all computationally expensive analysis parts will have already been performed. For example, in case of the higher order approach, as proposed by Jia and Harman [14], all required mutants executions will have been performed at the first steps of the higher order mutants construction. Thus, the fault localization expenses will be negligible. Further, it should be mentioned that equivalent mutants do not pose any problem to the localization approach. Since these mutants are not killed, they are ignored by Ochiai calculation formula. Therefore, they can safely be discarded from the employed mutant set along with those killable mutants that were not killed by the utilized test cases. These actions are performed at the testing phase.

VII. Related Work

There are a relatively large number of fault localization approaches appearing in the literature. Here a brief discussion on some of them is given.

As it has already described, Jones et al. [8] developed the Tarantula method and Abreu et al. [7] introduce the Ochiai formula. Both these advances were utilized by the propose mutation-approach in order to include mutants. Additional approaches employing different program elements such as program branches and definition use pairs are the ones of Marsi [11], Santelices et al. [6] and Yu et al. [12]. Marsi [11] concluded that branches and definition use pairs are more effective than statements. However, Santelices [6] showed that there is no approach that performs better in all cases, hence, proposing a combination of methods. Based on their results, Yu et al. [12] proposed a different combination approach. Similarly, Wong et al. [5] proposed a set of coverage-based heuristics able to improve the effectiveness of Tarantula.

Other related approach not using information is the Delta Debugging method [2], [3], [4]. This method recognizes and isolates input parts responsible for failures [4], recognizes chains of program states that lead to the failure [3] and links these chains with the faulty code. Recently, Burger and Zeller [32] proposed a combination of delta debugging and program slicing techniques to aid the whole debugging process. Their approach produces a test case that involves the minimum number of objects and method calls related to an examined failure, thus, assisting the programmers in reasoning about the failure. Jeffrey et al. [25] proposed a value profiling method to localize program faults. In this approach, variables at each program statement are assigned with different to the original execution values. These value replacements help identify the faulty statements by observing the programs' outputs.

Baudry et al. [20] suggested a different approach to assist fault localization. Instead of using existing tests in the localization process, to generate and optimize the whole suite according to an introduced criterion. This criterion was shown to be able to improve the fault diagnosis accuracy. This approach is somehow orthogonal to the one proposed in this paper. If tests can be optimized with respect to fault localization then the proposed approach will be assisted to provide better results. A different mutation-based debugging approach is that of Debroy and Wong [33]. In [33] it is suggested to use mutants in order to fixing faults. This approach operates after the localization process differing from the present one. Considering this approach it can be argued that mutation analysis is capable of supporting testing, fault localization and fault correction processes.

VIII. Conclusions and Future Work

Supporting both testing and localization activities with mutation analysis is the key-contribution of this paper. Mutants can be used first for guiding the production of test cases, therefore identifying program failures, and then in assisting the debugging process.

The work presented in this paper provides a number of insights to the fault diagnosis research. Primary, ***it shows that mutants can be utilized for the efficient localization of "unknown" faults***. Further, it validates this hypothesis in the cases of Block, Branch, Mutation adequate test sets and a relatively large and comprehensive test suite leading to the conclusion that ***mutants are suitable for both testing and debugging processes***. Finally, the practical use of mutation-based approach via mutant sampling was also investigated and showed that the ***mutation-based fault localization method is still efficient while used in a degraded situation, using few mutants***.

The major contributions made by the present paper can be summarized on the following points:

- The application of mutation analysis in assisting the fault localization process. The obtained results suggest that mutation-based fault localization is an effective technique, able to locate approximately ***90% of the utilized program faults by investigating at most 10% of the program code***.
- An experimental comparison of fault localization based on Block, Branch and Mutation based tests. Compared to the other criteria, ***mutation-based test cases***

significantly improve the effectiveness of the fault localization approaches.

- An empirical investigation of the practical usage of mutation-based fault localization, with reduced cost by mutant sampling. ***With only 10% of the mutants, the approach is still more effective, statistically significant, than the statement-based approach.***

Issues for further investigation include the use of weak mutation [22] as an alternative to mutation-based fault localization. Since weak mutation has been shown to be quite efficient [34] with respect to the test execution phase, utilizing it seams to be worthwhile. Additionally, the combination the proposed approach with the mutants' impact [16] may lead to reason about the located defects.

REFERENCES

[1] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," SIGPLAN Not., vol. 41, pp. 169-180, 2006.

[2] H. Cleve and A. Zeller, "Locating causes of program failures," in Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005, pp. 342-351.

[3] A. Zeller, "Isolating cause-effect chains from computer programs," in Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, USA, 2002, pp. 1-10.

[4] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," IEEE Trans. Softw. Eng., vol. 28, pp. 183-200, 2002.

[5] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," J. Syst. Softw., vol. 83, pp. 188-208, 2010.

[6] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 56-66.

[7] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "On the Accuracy of Spectrum-based Fault Localization," in Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007, pp. 89-98.

[8] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in Proceedings of the 24rd International Conference on Software Engineering, 2002, pp. 467-477.

[9] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in Proceedings of the 20th international Conference on Automated software engineering, 2005, pp. 273-282.

[10] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: statistical model-based bug localization," SIGSOFT Softw. Eng. Notes, vol. 30, pp. 286-295, 2005.

[11] W. Masri, "Fault localization based on information flow coverage," Software Testing, Verification and Reliability, vol. 20, pp. 121-147, 2010.

[12] K. Yu, M. Lin, Q. Gao, H. Zhang, and X. Zhang, "Locating faults using multiple spectra-specific models," in Proceedings of the Symposium on Applied Computing, 2011, pp. 1404-1410.

[13] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the Accuracy of Fault Localization Techniques," in Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009, pp. 76-87.

[14] Y. Jia and M. Harman, "Higher Order Mutation Testing," Inf. Softw. Technol., vol. 51, pp. 1379-1393, 2009.

[15] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 99, 2010.

[16] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, 2010, pp. 45-54.

[17] J. Yue and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," in Testing: Academic & Industrial Conference, 2008, pp. 94-98.

[18] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in Proceedings of the symposium on The foundations of software engineering, 2009, pp. 297-298.

[19] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," IEEE Trans. Softw. Eng., vol. 32, pp. 608-624, 2006.

[20] B. Baudry, F. Fleurey, and Y. L. Traon, "Improving test suites for efficient fault localization," in Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006, pp. 82-91.

[21] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, vol. 11, pp. 34-41, 1978.

[22] A. J. Offutt and R. H. Untch, "Mutation 2000: uniting the orthogonal," in Mutation testing for the new century, 2001, pp. 34-44.

[23] 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, pp. 99-118, 1996.

[24] M. Papadakis and N. Malevris, "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies," in Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, 2010, pp. 90-99.

[25] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in Proceedings of the 2008 international symposium on Software testing and analysis, 2008, pp. 167-178.

[26] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," Empirical Softw. Engg., vol. 10, pp. 405-435, 2005.

[27] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in Proceedings of the 16th international conference on Software engineering, 1994, pp. 191-200.

[28] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 60-71.

[29] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. d. S. Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero, "Proteum: a family of tools to support specification and program testing based on mutation," in Mutation testing for the new century, 2001, pp. 113-116.

[30] X. Zhang and R. Gupta, "Whole execution traces and their applications," ACM Trans. Archit. Code Optim., vol. 2, pp. 301-334, 2005.

[31] M. R. Lyu, J. R. Horgan, and S. London, "A coverage analysis tool for the effectiveness of software testing," in Software Reliability Engineering, 1993. Proceedings., Fourth International Symposium on, 1993, pp. 25-34.

[32] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 221-231.

[33] V. Debroy and W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs," in Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 65-74.

[34] 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, pp. 691-723, 2011.