SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1509

Metallaxis-FL: mutation-based fault localization

Mike Papadakis*,† and Yves Le Traon

Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg

SUMMARY

Fault localization methods seek to identify faulty program statements based on the information provided by the failing and passing test executions. Spectrum-based methods are among the most popular ones and assist programmers by assigning suspiciousness values on program statements according to their probability of being faulty. This paper proposes Metallaxis, a fault localization approach based on mutation analysis. The innovative part of Metallaxis is that it uses mutants and links them with the faulty program places. Thus, mutants that are killed mostly by failing tests provide a good indication about the location of a fault. Experimentation using Metallaxis suggests that it is significantly more effective than statement-based approaches. This is true even in the case where mutation cost-reduction techniques, such as mutant sampling, are facilitated. Additionally, results from a controlled experiment show that the use of mutation as a testing technique provides benefits to the fault localization process. Therefore, fault localization is significantly improved by using mutation-based tests instead of block-based or branch-based test suites. Finally, evidence in support of the methods' scalability is also given. Copyright © 2013 John Wiley & Sons, Ltd.

Received 3 September 2012; Revised 21 April 2013; Accepted 4 August 2013

KEY WORDS: debugging; mutation analysis; fault localization

1. 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 defective program parts given 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 activity.

When considering testing and fault detection, more than two decades of experiments on mutation testing have demonstrated that detecting artificial defects (e.g. seeded using mutation operators) allows effective detection of unknown, real ones, compared with more classical test selection criteria (e.g. based on code coverage). Test cases generated using mutations are good candidates for finding real faults [1, 2].

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 (a) whether *mutants could provide sufficient guidance for localizing known but not located faults*, that is, faults that have already been detected by some tests but not located in the code, and (b) whether *test cases that are able to kill mutants would enable accurate fault localization*.

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

*Correspondence to: Mike Papadakis, Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg.

†E-mail: Michail.Papadakis@uni.lu

approaches, such as dynamic slicing [3], produce a set of program statements that influence the output of the program. Delta debugging [4–6] isolates the causes of program failures by examining the state differences between passing and failing program executions. Other techniques, usually referred to as coverage based [7, 8], 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 program statements [9–11], program branches [8, 12], program du-pairs [13] and possible combinations of them [8, 14]. Empirical evidence has shown that coverage-based fault localization approaches can be very effective and helpful [11, 15] in diminishing the debugging effort.

Among the various coverage elements considered as 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 researchers. This might attribute to the general belief that mutation testing is quite expensive and cannot scale [16]. However, recent advances have shown that mutation testing can be practical [16, 17] and can be applied on real-world applications [16–18]. Many efficient and scalable mutation testing tools such as the MiLu [19] and Javalanche [20] have been built with promising results. Furthermore, integrating mutation analysis both for testing and fault localization activities may keep the fault diagnosis expenses at a low level.

Mutation analysis works by introducing defects named mutants in the program under analysis. Mutation analysis relies on the assumption that most of the mutants form ‘realistic’ faults, even if artificially seeded. Several empirical results, such as those of Andrews *et al.* [1], provide evidence that this assumption is reasonable. Therefore, the following question can be positioned: If mutants’ detection results in revealing ‘unknown’ faults, is the location of mutants able to assist with the localization of these faults?

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 reconcile testing and diagnosis activities, which are usually targeting different objectives (fault detection and fault localization [21]). Minimizing the effort of the testing process requires the minimization/prioritization of the test cases, whereas minimizing the fault localization effort requires the maximization of the information provided by test execution.

This paper proposes Metallaxis-FL, a fault localization approach based on mutation analysis. Metallaxis is the Greek word for mutation; in English, it is used in the context of describing moth species basically when they change from worm to moth. The present paper forms an extension of the authors’ previous work on fault localization [22] and aims at investigating (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) whether mutation alternatives such as mutant sampling can be utilized to support the fault localization activity. The aforementioned questions were explored on a set of 11 benchmark programs by using their accompanied test suites and faulty versions.

The remainder of this paper is organized as follows. Sections 2 and 3 present the underlying concepts and detail the proposed approach. Its evaluation set-up along with empirical results is described in Sections 4 and 5, respectively. Sections 6 and 7 discuss the proposed technique and its relation to the literature. Finally, Section 8 identifies some threats to validity, and Section 9 concludes the paper and reports some possible future directions.

2. MUTATION TESTING AND FAULTY STATEMENTS

Provoking program failures forms the primary aim of the testing process. Programmers, 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 the fault. Testers usually utilize adequacy or coverage criteria in order to assist them with the testing process. Fault diag-

DOI: 10.1002/stvr

nosis techniques prioritize the program places in order to help testers locating faults. This section summarizes the aforementioned concepts, which underlie the work presented in this paper.

2.1. *Code coverage*

Software testing process consists in checking the software's behaviour by executing a set of test cases. Test adequacy criteria (also called coverage criteria) are defined to help testers select a set of tests that is small but representative of the whole possible cases. This is achieved by possessing the requirement on the selected test cases to cover, that is, execute some specific program elements called *test elements*. In view of this, different program elements give rise to different criteria, that is, by setting them as test elements. Program elements such as the program lines, basic blocks, decisions and du-pairs are some of the most popular ones used for testing. The present paper considers block and decision criteria that require from the test cases to cover-execute all program blocks and decisions. Testing thoroughness with respect to a test criterion, here referred to as score or coverage level, is measured on the basis of the ratio of the test elements exercised by the test cases to the total number of elements required by the test criterion.

2.2. *Mutation analysis*

Mutation analysis is a fault-based technique. Thus, it introduces some defects named *mutants* in the program under analysis. Mutants are produced on the basis of 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 behaviour 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 outputs different from the original program version, whereas live are the unkilled mutants.

Mutation is based on the hypothesis of the 'competent programmer', that is, the assumption that programmers produce programs that are nearly 'correct' [23] and the 'coupling effect' [23]. The coupling effect states '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 exist: the original one, say O, and the mutated one, say M. If *making only one syntactic change to O produces M,* it is called first-order mutant. Otherwise, it is called a higher-order mutant [16]. This paper considers only first-order mutants. Mutation can be used as a test adequacy criterion and referred to as mutation testing. This is accomplished by using mutants as test elements. Thus, the ability of test cases, say t, to distinguish the output of the mutated version from the original program version is assessed. 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 classical criticism of mutation is its high computation cost. Because a vast number of mutants have 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' [24] and the 'selective mutation' [25]. 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 considered. Empirical evidence [25, 26] has shown that both of the aforementioned approaches are capable of constructing high-quality test data. More details about mutation and its alternatives can be found in [17, 24].

3. RANKING STATEMENTS USING CODE COVERAGE AND MUTATION ANALYSIS

This section discusses the use of coverage and mutant test elements to assist the fault localization process. We call an 'un-located fault' a fault that has been detected by at least one test case but that has still to be located.

DOI: 10.1002/stvr

Table I. The Ochiai formula.

$$\mathit{Suspiciousness}(e) = \frac{\mathit{failed}(e)}{\sqrt{\mathit{totfailed} * (\mathit{failed}(e) + \mathit{passed}(e))}}$$

where *totfailed* is 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.

3.1. Coverage-based fault localization

Research on fault localization suggests that utilizing the execution traces of the employed test suites can help in performing this process. These approaches referred to as coverage based [8] record the executed-covered code elements of the passing and failing test cases. The main idea is that code elements executed by failing test cases are responsible for the failure. Thus, for each of the considered program elements, a suspiciousness value is assigned. This value approximates the probability that a program statement is faulty, and it is calculated on the basis of the frequency that program elements appear in the failing and passing program executions. Then, the programmer is assisted to find a fault by inspecting these highlighted elements on the basis of a decreasing suspiciousness order, that is, starting from the most suspicious one to the least one.

One of the most popular coverage-based methods is Tarantula [10]. The 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]. Abreu *et al.* [9] investigated this issue and concluded that a similarity coefficient named *Ochiai* was the most effective one. Santelices *et al.* [8] also report that their experiments supported the use of this formula; hence, they used it in their approach. The Ochiai suspiciousness calculation formula for a code element e is presented in Table I. Its application is straightforward once the program execution traces are available, by measuring the number of failing and passing test cases that cover each one of the program statements. Then, for all the program statements, a suspiciousness value is calculated on the basis of the Ochiai formula. Finally, the program statements are ranked according to their suspiciousness, and they are reported to the user.

3.2. Mutation-based fault localization

The mutation-based fault localization approach identifies suspicious mutants and uses their location in order to identify the un-located fault statements. The identification of suspicious mutants is based on their behaviour. A mutant *M1 is said to have the same behaviour with another mutant M2 if M1 and M2 are killed by the same test cases*. The degree of similarity on the test cases that kill the mutants M1 and M2 define the *behaviour similarity* of these mutants. Generally, the proposed approach is motivated by the following two observations:

- Mutants-faults located on the same program statements frequently exhibit a similar behaviour.
- Mutants-faults located in diverse program statements exhibit different behaviours. For a 'hard-to-kill' mutant, a test case that kills it is capable of killing the mutants located on the same statement.

Consider a scenario where the program under test contains an un-located fault. On the basis of the aforementioned observations, this fault behaves similarly to seeded faults applied at the same statement. At the same time, faults seeded on other statements behave differently to the un-located one. Then, within this assumption, the 'un-located and seeded faults exhibit similar behaviours'; the identification of an un-located fault may be obtained thanks to a mutant fault at the same location. Section 3.2 provides an example of the proposed approach illustrating the aforementioned scenario.

The intuition behind the proposed approach is this implicit link between the behaviours of un-located faults with other local mutants. The behaviour of the un-located faults is identified on the basis of the passed and failed test cases. Mutants' behaviour is identified on the basis of the utilized test cases that achieve to kill or not the considered mutants. The similarity between these two behaviours establishes the sought implicit link between the un-located faults and mutants. This can be performed straightforwardly by using the ways in which existing fault localization techniques quantify the similarity between the structural code elements (such as statements or decisions) and the un-located faults behaviour. Therefore, by measuring the number of killed mutants by the passing and failing test executions, one can have an indication about the suspiciousness of those mutants. This can be computed by applying the Ochiai formula (Table I) with elements (e) representing the killed mutants. Following the lines of the spectrum fault localization approaches, killed mutants are treated as covered elements (e), while the live ones are ignored, that is, treated as uncovered elements. Hence, all the introduced mutants (e), regardless of the statement they belong to, are assigned with a suspiciousness value.

The Metallaxis 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 to the program statements. Because mutants' location is known, the suspiciousness values computed for the mutants can be assigned to their respective statements. However, because most of the program statements involve many mutants, each one having its own suspiciousness, which is the suspiciousness value of a program statement? In other words, there is a need to assign a suspiciousness value to a program statement on the basis of the suspiciousness values of the mutants that are belonging to this statement. In this paper, the suspiciousness of a statement is equal to the maximum suspiciousness value of its respective mutants. This is in line with the fault localization methods, where the most suspicious coverage elements have to be investigated first. Furthermore, the intuition of the proposed approach is that the similarities between the behaviours of the faults and the mutants are attributed to their location. Depending on the utilized operators, it is possible that some statements might not have any mutants. These statements are assigned with the worst suspiciousness value (the number of program statements). This practice ensures that these statements will be among the last ones to be inspected reflecting the inability of the approach to highlight the faulty statement. Generally, the assignment of suspiciousness values on non-mutated statements can be performed on the basis of the lines suggested by Santelices *et al.* [8]. Such statements (without suspiciousness) can be mapped with the values of the statements that are data or control dependent [8]. However, the need for this practice is not crucial in our case, because mutants operate on most of the program statements.

3.3. *An illustrative example*

Consider the example of Figure 1, which illustrates the use of statements and mutants in localizing faults. This example shows

1. how the examined fault localization approaches work and
2. a concrete scenario of mutant-fault localization using different types of mutants.

To demonstrate how the examined fault localization approaches work, consider the example of Figure 1, which has been taken from the work of Santelices *et al.* [8]. This example, program *mid*, involves two faults named as Fault 1 and Fault 2. Fault 1 (localized in statement 3) is due to extra code fragments ($y < z \rightarrow y < z - 1$), and Fault 2 (localized in statement 7) is an assignment expression error ($m = x \rightarrow m = y$). The example program has 13 statements (column Statements) and is tested with six test cases (top of the columns Tests 1–6). Test columns record the covered test elements, that is, statements and mutants, by the respective test cases (denoted with 'X' for executed statements and with '$\surd$' for the killed mutants). 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 cover-kill each program statement-mutant. The column labeled as 'Suspiciousness' record the suspiciousness scores (calculated by the Ochiai formula, Table I) per statement and mutant. The column labeled as 'Rank' record the respective ranking for all the program statements. The upper part (above the black line) of the figure corresponds to the statement-based approach, and the lower part of the figure corresponds to the mutation-based approach.

DOI: 10.1002/stvr

Fault1: Statement 3 (if (y < z - 1))

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

Fault2: Statement 7 (m = y)

	Mutants	Statements	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	#Passed	#Failed	Suspiciousness	Rank
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	X	X	X	X	X	X	5	1	0.41	7
m = z;		2	X	X	X	X	X	X	5	1	0.41	7
if (y < z)		3	X	X	X	X	X	X	5	1	0.41	7
if (x < y)		4	X	X			X	X	3	1	0.50	3
m = y;		5		X					1	0	0.00	13
else if (x < z)		6	X				X	X	2	1	0.58	2
m = x;		7	X					X	1	1	0.71	1
else		8			X	X			2	0	0.00	13
if (x > y)		9			X	X			2	0	0.00	13
m = y;		10			X				1	0	0.00	13
else if (x > z)		11				X			1	0	0.00	13
m = x;		12							0	0	0.00	13
return m;		13	X	X	X	X	X	X	5	1	0.41	7
}												
mid(**int** x, **int** y, **int** z){												
int m;		*1*										*13*
m = z;	*M1. z → z+1*	*2*				√	√		*2*	*0*	*0*	*13*
	M2. z → z-1					√	√		*2*	*0*	*0*	
if (y < z)	*M3. y → y+1*	*3*		√			√		*2*	*0*	*0*	*13*
	M4. y → y-1								*0*	*0*	*0*	
	M5. z → z+1								*0*	*0*	*0*	
	M6. z → z-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*	*0*	*0*	*13*
	M12. y → y-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*	*0.71*	*1*
	M18. y → y-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*	*0*	*0*	
	M21. y → y+1				√				*1*	*0*	*0*	
	M22. y → y-1								*0*	*0*	*0*	
m = y;	*M23. y → y+1*	*10*			√				*1*	*0*	*0*	*13*
	M24. y → y-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*	√	√	√	√	√	√	*5*	*1*	*0.41*	*2*
	M32. m → m-1		√	√	√	√	√	√	*5*	*1*	*0.41*	
}			P	P	P	P	P	F				

Figure 1. Fault localization example using program statements and mutants. The upper part corresponds to a statement-based approach, while the bottom part corresponds to the mutation-based one. The symbols X and √ denote that the test cases (columns) cover-kill the specified statement (rows).

Fault 1 (localized at statement 3) is detected by test cases 2 and 5 (bottom of the columns Tests 1–6). This fault is ranked in the sixth position by the statement-based (upper part of the figure) fault localization method. The total number of test cases that fail is two and that passed is four. Because the two failed test cases and the four passed test cases cover this statement, on the basis of the Ochiai formula, its suspiciousness is calculated as 0.58. Similarly, the suspiciousness of statement 9 is calculated as 0.71 because two failed and two passed test cases cover this statement. The mutation-based approach precisely localized the fault (ranked in the first position) based on the relational mutant operator, that is, it changes the instance of relational operators with the other ones.

The utilized mutant elements are demonstrated in the column Mutants (left part corresponding to Fault1), and they are named as $M1$–$M35$. Because mutant $M1$ is killed by the two failing test cases and not by any of the passed, it has a suspiciousness value of 1.0. Thus, the statement of M1 is reported as the most suspicious one.

Fault 2 (localized at statement 7) is detected by one test case, and it is ranked in the first position by both fault localization methods. This statement is covered by one failing and one passed test cases; thus, the statement-based approach assigns a suspiciousness value of 0.71. The mutation-based approach localizes this fault by using the numerical constant increment and decrement mutants, that is, it adds and subtracts a constant value to a program's variable. The mutant elements used are presented in the column Mutants (right part corresponding to Fault2), and they are named as $M1$–$M32$. On the basis of the $M17$ and $M18$ mutants, which are killed by one failing and one passed test cases, the faulty statement is ranked in the first position, and it has a suspiciousness value (0.71).

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, if faulty statements appear at a higher position in the ranked order, this is preferable because less effort is required to find the error.

Conclusively, Figure 1 demonstrated a scenario of applying spectrum-based fault localization methods. By using two faults, the process of assigning suspiciousness values to program statements based on both statement coverage and mutation analysis was also presented.

4. EXPERIMENTAL STUDY

This section discusses the empirical set-up 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 set-up and analysis is provided.

4.1. Experimental objectives

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 randomly selected mutant sets? Comment: in this study, random sampling of 10%, 20%, 30%, 40% and 50% mutant sets were used.
- RQ4: How effective is the mutation-based fault localization approach when it is applied on large software subjects?

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 experiencing a failure or if they should produce some additional tests first. Additionally, because mutation-based fault localization relies on the killed mutants, it is expected to be less effective when these tests are incapable of killing many mutants. Therefore, RQ2 will give an answer whether mutation-based localization approach is worthwhile when employing a basic testing approach such as block or branch coverage. Research questions RQ3 forms an attempt towards dealing with the computational demands of mutation analysis. By facilitating only few mutants, a practical answer to the computational cost of the method can be given. An investigation on the localization ability of these attempts is carried out. Finally, RQ4 explores the scalability of the mutation-based approach on larger programs. By showing that Metallaxis works well on large programs, the practicality of the approach is established.

4.2. Subject programs, faults and test suite pools

The conducted experiment employed two sets of benchmark programs. The first set is composed of relatively small subjects, and it was used in order to answer RQ1–RQ3. The second one is composed of large subjects, and it was used for exploring the scalability of the mutation-based approach (RQ4).

The first benchmark set is known as the Siemens suite and has been widely used in mutation testing and fault localization experiments (e.g. [4, 8, 11, 12, 14, 27]). In the rest of the paper, this program set is referred as the *Siemens* suite or as the *benchmark set 1*. The suite is composed of seven programs written in C and is accompanied by test suite pools and a set of 132 faults. These programs were chosen because of their widespread use in the literature on the one hand and their availability along with their accompanied tests and fault sets from the Software-artifact Infrastructure Repository (SIR) at the University of Nebraska–Lincoln [28] on the other. During the experimentation, one fault was excluded from the considered set, because it did not result in any execution failure, a mandatory requirement of the fault localization methods. This action was also taken in other similar studies (e.g. [8, 11, 27]). Additionally, to validate further the proposed approach, the present study employs another 100 faulty versions per subject program. These faulty program versions are produced by randomly selecting some mutants and are denoted as *mutant-faults*. Mutant-faults as alternatives to faults in examining the effectiveness of fault localization techniques were also used in the study of Baudry *et al.* [21]. The validity of this practice was later investigated by Ali *et al.* [15] and found ‘no evidence to suggest that the use of mutants for this purpose is invalid’.

The program suite was initially employed in an empirical study by Hutchins *et al.* [29] for comparing various structural testing criteria. Later, it was extended and adapted appropriately from other researchers to support their experiments [30]. According to Hutchins *et al.* [29], the accompanied set of faults was manually produced by various researchers with the intention of introducing realistic faults. The accompanied test suites were produced on the basis of 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 [30].

The second set of benchmarks, referred to as *benchmark set 2*, is composed of four programs named as the Gzip, Space, Grep and Flex. These programs have also been used in fault localization studies such as [31, 32] and can be obtained from the SIR repository. Gzip functions as a utility for compressing files, Space as an interpreter for an array definition language, Grep as a utility for text searching and Flex as a lexical analyser. For the present study, the faulty versions of the Space (all versions), Gzip (v1), Flex (v2) and Grep (v3) programs were used. Unfortunately, not all the faults were detected by any test from their accompanied test suites when executed in the experimental environment. This is also reported by other studies using the same programs (e.g. [31, 32]). Therefore, Gzip has 7, Grep has 8, Flex has 13 and Space has 34 faults that can be detected. For the case of Space among the 34 detected faults, 12 of them were randomly chosen among those handled by Gcov, that is, some faults resulted in program failures that prevent Gcov to collect their traces. Thus, a total number of 40 faults were considered. Table II records details about the program lines of code, the size of the test pools, the number of faults and the number of mutant-faults per program for both of the utilized benchmark sets.

4.3. Utilized tools and implementation details

The present study used the Proteum[‡] mutation testing system by Maldonado *et al.* [33] in order to support the mutation analysis process. Proteum employs mutant operators specially designed for the unit test of C programs, implemented as suggested by the Agrawal *et al.* [34] study. To gather

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

DOI: 10.1002/stvr

Table II. Description of the selected subjects.

Benchmark set	Program name	Lines of code	Whole test suite	Number of faults	Number of mutant-faults
1	Schedule	296	2650	9	100
	Schedule2	263	2710	10	100
	Tcas	137	1608	41	100
	Totinfo	281	1052	23	100
	Printtokens	343	4130	7	100
	Printtokens2	355	4115	10	100
	Replace	513	5542	32	100
2	Gzip	6576	214	16	—
	Space	9564	13585	38	—
	Grep	13341	809	18	—
	Flex	14120	567	20	—

the required tracing information, a prototype has been implemented on top of the Wet [35] framework in the same lines utilized in [14]. 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 on the basis of their line numbers. The prototype implements both the statement-based and mutation-based approaches utilizing the Ochiai formula (given in Table I). Unfortunately, the use of Wet framework restricts the scalability of the proposed approach. To accomplish the scalability study, RQ4, executable statements and traces were collected using Gcov.[§] Gcov is a widely used profiler, also employed in some fault localization studies such as [31].

The ATAC [36] coverage tool was used for the selection of the Block and Branch test sets from the accompanied test pools and Proteum [33] for the mutation ones. These tools have also been used in software testing experiments (e.g. [1, 17, 26]). Details about the test selection process are given in the following subsection.

4.4. Experimental regime

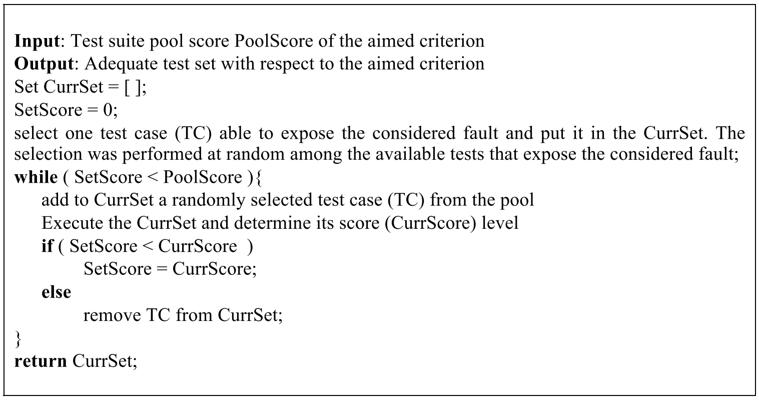
The following experiment was set to address the stated RQs.

Regarding RQ1–RQ3, all the Siemens suite subjects, benchmark set 1, (including the faulty ones) were executed with all the available test cases in order to record the passing and failing executions of the whole 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. The study of RQ1 and RQ3 was based on these results.

With respect to the mutation-based approach (examined by RQ1), the entire set of utilized mutants were generated, compiled and executed against the whole 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 (investigated by RQ3) was performed by selecting and generating at random only a percentage of the whole set of mutants. Five different sampling ratios were studied (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. 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[¶] 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 per studied fault. The test sets were constructed from the available test suite pool using the procedure of Figure 2. In this figure, the term score refers

[§] Gcov is a GNU code coverage tool part of GCC.

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



```
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 CurrSet;
}
return CurrSet;
```

Figure 2. Test selection procedure.

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 on the basis of random selection from the available test pool. Each one of these tests was able to expose its respective fault. 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.

Regarding the scalability of the method, RQ4, the four larger programs, benchmark set 2, were used. For each of the 40 faults considered, a maximum of 10 test cases, five failing and five passing, were employed. This provided a good mix of passing and failed test cases. To this end, per examined fault, five failing test cases were randomly sampled. In some cases, less than five failing tests exist. In these cases, all the available failing test cases were used. Similarly, five passing tests were randomly sampled from the respective test suite pools. In order to provide a challenging situation for the fault localization approach, the selection of the passing test cases was performed by sampling among the cases that execute the faulty statement and are passing. In the lack of five such cases, the sample was extended, up to five tests, by randomly choosing passing tests.

Only the executable statements were ranked in the present experiment because of the function of the developed tool (Section 4.3). Additionally, to avoid repeating the mutation analysis process for all considered faults, the localization process was performed on the original (correct) program versions by treating the ‘correct’ program as being faulty one and the faulty as being ‘correct’. This restriction was applied only on the programs of the benchmark set 1 in order to complete the experiment with reasonable resources (vast resources are typically needed by mutation analysis). Thus, it helped not to repeat the process for each considered fault (the number of faults considered by the benchmark set 1). It is noted that on the benchmark set 2, the fault localization process was performed on the faulty programs.

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 Figure 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. [8, 11, 27]).

Comparing the localization 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.* [11] in evaluating the Tarantula fault localization system. The ‘score’ measures the percentage of executed program statements that need not be checked 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 on the basis of the order specified by the fault localization tool. Along these lines, the score value is calculated on the basis of the following formula:

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

In the aforementioned 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. The effectiveness comparisons between the studied methods were performed using the Mann–Whitney U-test. The Mann–Whitney U-test examines whether one of the compared approaches tends to have higher values than the other one, and it is a nonparametric statistical hypothesis test. Thus, it does not make any assumptions about the underlying populations.

Lastly, some additional concerns were made about the utilized faults and their localization. To avoid any bias introduced by the mutant-faults and the mutation-based fault localization approach, the mutant-faults were removed from the considered mutant set utilized by the localization method. This practice is opposed to the mutation-based fault localization because it decreases the number of mutants that it uses and hence its effectiveness. 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 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.

Conclusively, to address RQ1, the score measures according to both the mutation-based and statement-based methods were obtained and compared. This was accomplished with the use of the whole test suite for all the studied faults and mutant-faults. To address RQ2, the effectiveness of the fault localization methods was analysed on the basis of the selected set of test cases. The experiment considers in total 1310 test sets (131 faults × 10 test sets) and 7000 (700 mutant-faults × 10 test sets) test sets per utilized testing criterion. To address RQ3, results were derived on the basis of the 50 randomly selected mutant sets (10 sets per sampling ratio). For each one of these mutant sets, its respective score values were evaluated using the whole test suite for all the studied faults and mutant-faults. Finally, considering RQ4, the effectiveness values, on the benchmark set 2, is reported and analysed.

5. EXPERIMENTAL RESULTS

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

5.1. Effectiveness evaluation – (RQ1)

The effectiveness results of mutation-based and statement-based approaches, with respect to RQ1, are summarized in the graphs of Figure 3. The obtained results are categorized on the basis of the assigned scores (this practice is also used in [14, 15, 27]) in the following categories: 99–100%, 90–99%, 80–90%, 70–80%, 60–70%, 50–60%, 40–50%, 30–40%, 20–30%, 10–20% and 0–10%. Tables III and IV in the columns 'StLoc whole-suite' and 'MutLoc whole-suite' record the percentage of faults (Table III) and mutant-faults (Table IV) found by the statement-based and mutation-based approaches per score range, respectively. These results are cumulatively depicted in the graphs of Figure 3. In these plots, y-axis records the percentage faults that are effectively localized, while the x-axis the percentage ranges of statements that does not need inspection by the programmer. It is noted that method curves (data points) appearing higher in the graphs reflect less effort by the programmer and thus 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 approach and only 0.44 with statement-based approach, by inspecting only a 10% of the programs' code.[‖] With respect to faults, mutation-based approach achieved to effectively

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

DOI: 10.1002/stvr

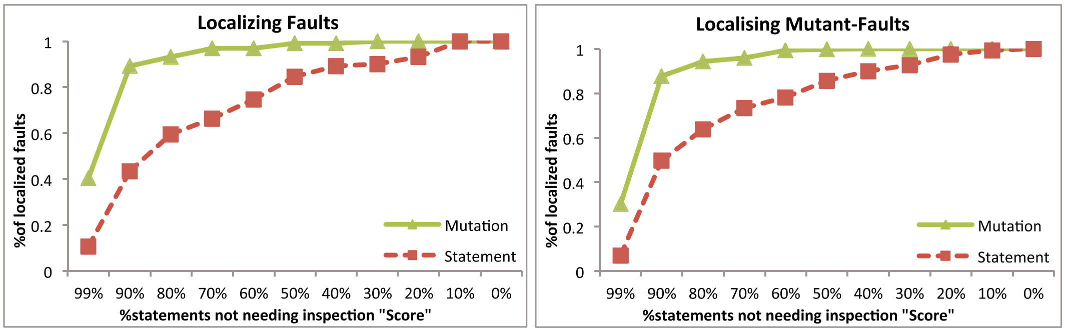


Figure 3. Effectiveness comparison of the mutation-based and statement-based fault localization methods using the whole test suite.

localize the *0.41, 0.89* and *0.93* of faults in the *99%, 90% 80%* categories, while the statement-based one *0.11, 0.44* and *0.60*, respectively. With respect to mutant-faults, mutation-based approach achieved to effectively localize the *0.30, 0.88* and *0.94* of mutant-faults, while the statement-based one *0.07, 0.50* and *0.64*, respectively. These results 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%* with respect to faults. Regarding mutant-faults, these values are *79%* and *95%* for the statement-based and mutation-based methods, respectively.

Figure 4 depicts the fault localization cost, that is, the percentage of statements that need inspection by the programmer in order to effectively localize the sought faults, per fault and mutant-fault. It can be observed, from this graph, that the cost for localizing faults with mutation-based approach is lower in most cases. Additionally, the difference is considerable for most cases. Regarding the sample of the 131 faults, mutation-based approach performed better in the 108 cases, worst in 17 cases and had equal effectiveness in six cases. An examination of these 17 cases reveals that in nine cases, this difference was less than 1%. Only in four cases the statement-based approach was better by more than 5% but no more than 11%. Considering the sample of 700 mutant-faults, mutation-based approach performed better in the 545 cases, worst in 113 cases and had equal effectiveness in 42 cases. These results suggest that whenever the statement-based approach achieves a better effectiveness, this difference is not so important.

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

Fault localization techniques are dependent on both the employed elements and the utilized 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, because the mutation-based fault localization method relies on the killed mutants, it is expected (intuitively) to experience 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 results concerning this issue, RQ2, are recorded in Tables III and IV and Figure 5. Tables III and IV record the ratio of the effectively localized faults at 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-based (StLoc) and mutation-based (MutLoc) fault localization methods. Similarly, Table V records the total average score values for the examined methods and test suites. A cumulatively view of these data is plotted in the graphs of Figure 5. Additionally, Tables III and IV record 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 localization approaches experience significantly better score values when utilizing test suites adequate with respect to mutation than those based on random, branch or block criteria. This situation holds in both the examined cases, regarding mutant-faults and faults.

Table III. Percentage of located faults with respect to score ranges 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	4.05	4.89	9.31	8.47	10.69	4.58	6.64	31.98	26.34	40.46
90–99	25.50	28.32	34.96	31.15	32.82	51.30	54.96	57.40	56.41	48.85
80–90	10.31	12.14	9.31	11.68	16.03	19.85	17.56	4.35	8.93	3.82
70–80	5.95	6.26	6.26	6.11	6.87	7.48	7.71	1.91	2.52	3.82
60–70	8.93	9.77	9.47	9.85	8.40	7.25	7.18	1.07	2.14	0.00
50–60	13.59	14.58	10.38	11.07	9.92	6.03	2.98	1.68	1.91	2.29
40–50	7.02	4.66	2.98	3.89	4.58	1.68	1.91	0.84	0.76	0.00
30–40	2.67	2.06	2.21	1.91	0.76	1.83	0.92	0.76	0.99	0.76
20–30	9.62	6.03	6.72	5.57	3.05	0.00	0.15	0.00	0.00	0.00
10–20	9.92	8.70	7.25	7.56	6.87	0.00	0.00	0.00	0.00	0.00
0–10	2.44	2.60	1.15	2.75	0.00	0.00	0.00	0.00	0.00	0.00

DOI: 10.1002/stvr

Table IV. Percentage of located mutant-faults with respect to score ranges 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	2.11	2.37	5.21	4.30	7.14	2.86	3.99	20.73	15.50	30.14
90–99	23.56	24.43	35.30	33.23	42.71	37.03	42.63	59.03	60.71	57.71
80–90	10.39	13.01	10.56	13.31	14.14	15.40	17.63	5.40	8.10	6.57
70–80	6.29	7.33	7.86	9.51	9.57	15.53	12.80	6.66	6.64	1.57
60–70	12.57	13.53	11.76	10.17	4.57	13.29	13.27	5.86	6.80	3.43
50–60	14.27	12.49	8.11	9.83	7.57	10.57	8.41	2.33	2.24	0.43
40–50	10.19	8.79	6.80	3.63	4.43	4.07	1.27	0.00	0.00	0.14
30–40	5.11	5.39	3.54	5.47	2.57	1.26	0.00	0.00	0.00	0.00
20–30	7.61	7.11	6.53	4.87	4.86	0.00	0.00	0.00	0.00	0.00
10–20	5.97	3.81	2.59	3.33	1.86	0.00	0.00	0.00	0.00	0.00
0–10	1.93	1.74	1.74	2.34	0.57	0.00	0.00	0.00	0.00	0.00

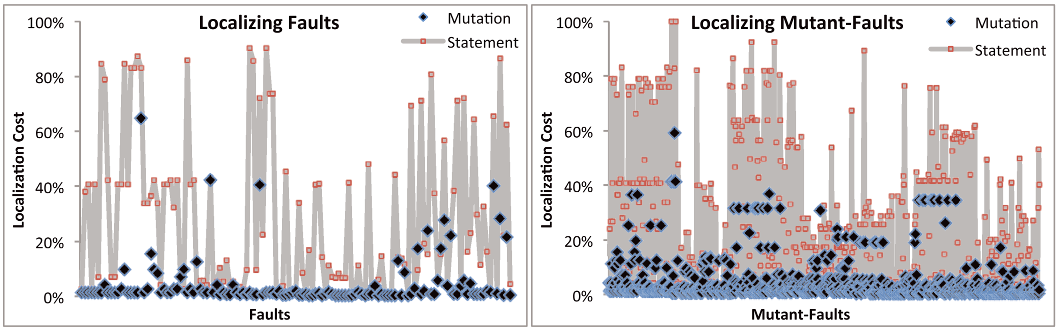


Figure 4. Fault localization cost per utilized fault and mutant.

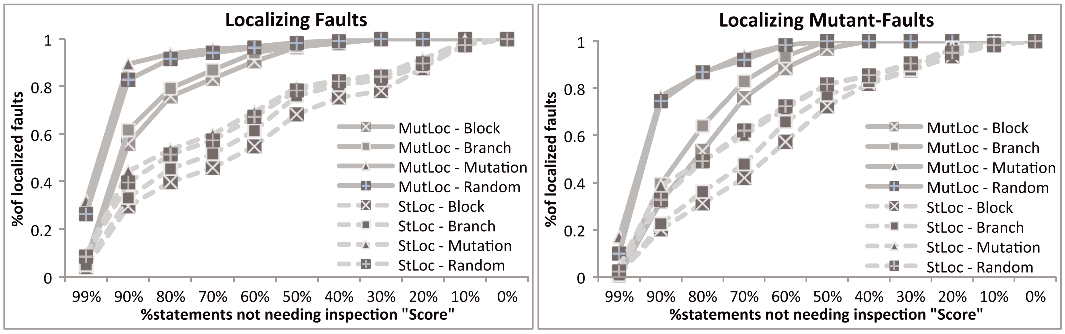


Figure 5. Effectiveness comparison of the mutation-based (MutLoc) and statement-based (StLoc) fault localization methods by utilizing block, branch and mutation and random test sets.

The most interesting finding of these results is that mutation-based localization approach outperforms the statement-based one in all cases, even when using block adequate test suites. Further, it was found that this difference is statistically significant** 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 4.2 for details). 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 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 in Table VI. From this table, it can be seen that there are considerably more tests with respect to mutation than with respect to branch or block. However, the results of Tables III and IV and Figure 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 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 of these hypothesis tests are recorded in Table VII and suggest that there is a statistically significant difference in the effectiveness of the mutation and the whole test sets. Therefore, there is room for improvement in the methods' effectiveness by producing additional test cases. The question that remains is how could that be achieved.

**Mann–Whitney U-test was used with a statistically significant difference of $p < 0.0001$.

Table V. Average score values of the faults and mutant-faults of the statement-based and mutation-based approaches with respect to the block, branch, random, mutation and the whole test suites.

	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
Faults (%)	63.34	67.52	72.55	70.70	76.61	85.41	87.38	94.80	93.31	95.42
Mutant-faults (%)	63.64	66.64	72.72	72.15	78.98	79.79	83.03	92.46	91.39	95.26

Table VI. 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

Table VII. Statistical comparison (p-values) of mutation, random and whole test suites.

	MutLoc (Mut-suite) versus MutLoc (Rand-suite)	StLoc (Mut-suite) versus StLoc (Rand-suite)	MutLoc (Mut-suite) versus MutLoc (whole-suite)	StLoc (Mut-suite) versus StLoc (whole-suite)
Faults	0.0000	0.0143	0.0465	0.2330
Mutant-faults	0.0000	0.0061	0.0000	0.0000

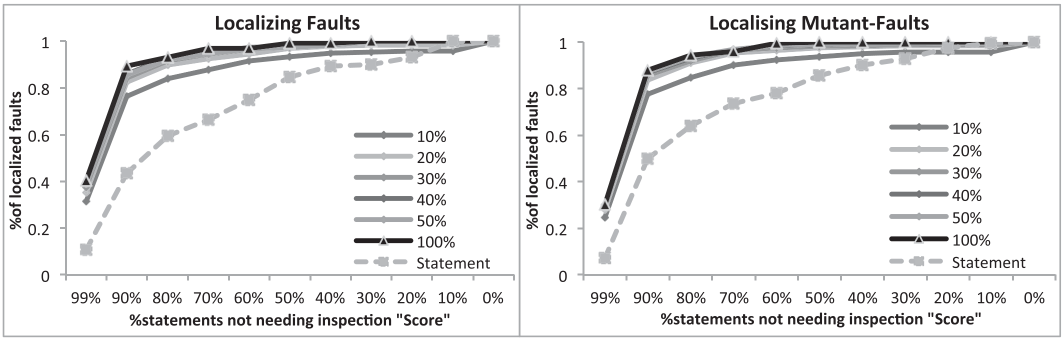


Figure 6. Mutant sampling approaches in assisting fault localization.

5.3. Mutant sampling evaluation – (RQ3)

Mutation analysis has been identified as a costly technique. To overcome its difficulties, various mutation alternative techniques have been proposed [17, 24]. The present study examines the use of mutant sampling in fault localization. Figure 6 presents the effectiveness results of the mutant sampling technique with sampling ratios 10%, 20%, 30% 40% and 50%. For evaluation reasons, Figure 6 also plots the results of the whole utilized mutant set (denoted as 100%) and the statement-based ones. The graphs of Figure 6 suggest that all the examined ratios experience loss in their effectiveness compared with the whole mutant set. In the case of 10%, this loss is more apparent than the rest of the utilized approaches, which have a similar effectiveness.

By statistically comparing (Table VIII) 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 ($p < 0.0001$ with respect to both faults and mutant-faults). On average, 10% mutant sampling achieved an effectiveness score of approximately 0.89 of the introduced faults, while the statement-based one only the 0.77 of them. Considering mutant-faults, on average, 0.89 and 0.79 score values were obtained by 10% mutant sampling and statement-based approaches, respectively. In view of this, it can be argued that mutation alternative methods can be effectively utilized to assist the fault localization process.

Table VIII. Statistical comparison (p-values) of mutation and mutant sampling.

	100% vs 10%	100% vs 20%	100% vs 30%	100% vs 40%	100% vs 50%
Faults	0.0002	0.0347	0.2308	0.5701	0.5429
Mutant-faults	0.0000	0.0071	0.2159	0.4721	0.7989

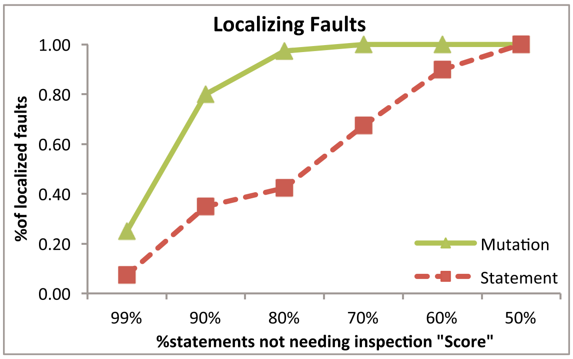


Figure 7. Effectiveness comparison of the mutation-based and statement-based fault localization methods for the benchmark program set 2.

5.4. *Scalability of the approach – (RQ4)*

This section explores the ability of the mutation-based approach to localize faults when it is applied on large subjects as stated by RQ4. To this end, the mutation-based and statement-based approaches were applied on the benchmark programs of set 2. The cumulative results of these approaches are summarized in the graph of Figure 7. Following the same lines as in the previously presented results, the y-axis records the percentage faults that are effectively localized. The x-axis records the percentage ranges of statements that do not need inspection by the programmer. Thus, the mutation-based approach achieved to effectively localize the 0.25, 0.80 and 0.98 of faults in the 99%, 90% and 80% categories, while the statement-based one 0.08, 0.35 and 0.43, respectively. These results 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 79%, while the mutation-based one 95%.

The most interesting finding of the conducted experiment is that the mutation-based approach is capable of localizing faults on large programs without being affected by their size. Although there is no fair way to compare the results of the two sets, due to the differences in the construction of the test suites, an inspection of Figures 3, 5 and 7 reveal the effectiveness similarities between the two program sets. The approaches have a similar trend on both sets and clearly show the superiority of the mutation-based one. The similarities are more evident by considering the average score values of the sets. Thus, considering the set 1, Table V, results in values within the range from 85% to 95% for the various considered test suites while the set 2 in 95%. Additionally, in Section 5.2, it was shown that there is a relation between test suite size and the fault localization effectiveness. Thus, a higher number of test cases results in a higher effectiveness on the fault localization. Because the set 2 contains much less test cases than the set 1 and the fault localization effectiveness of the mutation-based approach is almost the same in both sets, this is an indication that the mutation-based approach is more effective when applied on larger programs. In any case, it can be concluded that the mutation-based approach behaves similarly on both program sets and significantly better than the statement-based one.

Figure 8 forms a box-plot representation of the effectiveness of the examined approaches. The plot contains four boxes per approach, one for each considered program. The y-axis records the fault localization score. By comparing the two approaches on the basis of these results, it can be concluded that the mutation-based approach is clearly better than the statement-based one in all the examined programs. It is not only more effective but much more stable as well. A similar situation

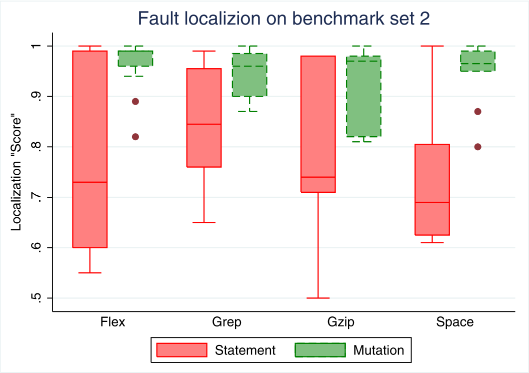


Figure 8. Effectiveness comparison of the mutation-based and statement-based fault localization methods for the benchmark program set 2.

appears on all the studied subjects. Finally, by inspecting Figure 8, it becomes evident that whenever the mutation method is more effective, this indicates a huge difference. In the opposite situation, that is, when the statement-based approach is better, the difference of the two methods is not important.

6. DISCUSSION

This paper presents the use of mutation analysis in assisting program debugging. Its key insight is the use of mutants for locating program faults. Generally, Metallaxis makes the assumption that mutants and faults located on the same program statement exhibit a similar behaviour. Evidence in support of this assumption was given by the reported experiments where mutant-faults were effectively localized using other mutants located on the same statements.

Generally, the provided empirical evidence strongly supports the conclusion that Metallaxis is able to locate program faults effectively. In fact, the results suggest that it significantly outperforms fault localization based on statement coverage. This difference is mainly attributed to the fact that when a fault is executed, it is not always manifested to a failure. Because coverage-based fault localization techniques try to find program statements that correlate with failures, they fail to identify faults that are frequently executed but not exposed. Mutants have the advantage of handling this case effectively. This is attributed to the so-called propagation requirement, that is, in order to be killed; discrepancies introduced by mutants must impact the programs' output. In practice, this makes a big difference because it does not treat each program statement equally. It actually treats program statements according to their sensitivity to impact the programs' output. On the basis of this requirement, mutants are able to simulate well the faults' behaviour [2]. The same idea applies on the automated oracle generation, as proposed by Fraser and Zeller [37], where important oracles are considered only those that detect-kill mutants. Here, it must be noted that easy to kill mutants, that is, killed by the most test cases that executed them, form substitutes to the coverage measures. Thus, the mutation approach is also capable of locating faults that are easy to expose when they are executed. Metallaxis has also some additional advantages over the statement-based approaches. This is its ability to capture the data flow information and therefore differentiate between program statements belonging on the same basic blocks. Hence, the approach is not dependent on the program structure, which usually results in assigning the same statement rankings (in the case of the statement-based approach) to many statements.

Fault localization is performed after the testing process, and thus, the proposed approach has many opportunities to reuse information from the testing process. Conversely, if mutation was employed during the testing phase, then most and perhaps all computationally expensive analysis parts (mutant executions) will have already been performed. For example, in case of the higher-order mutation, as proposed by Jia and Harman [16], in the case of automated oracle generation, as proposed by Fraser and Zeller [37], or in case of the equivalent mutant isolation, as proposed by Kintis *et al.*

DOI: 10.1002/stvr

[38], all required mutant executions will have been performed before the debugging process. 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. Because these mutants are not killed, they are ignored by the 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 also performed at the testing phase.

Finally, it is noted that the efficient introduction and execution of the sought mutants is a matter not addressed by the present paper and has been left open for future research. A possible solution could be the use of an incremental process. Thus, some mutants could be selected either at random or on the basis of some of their characteristics, that is, their kind or their suspiciousness value as computed by lightweight fault localization approaches. Then, their suspiciousness can be calculated, and the statements with score 1 are reported to the user. Thus, the programmer can start the debugging process before the whole process is finished. In case the fault is not found, then the process could continue with other mutants.

7. THREATS TO VALIDITY

Considering the results reported in the present paper, some threats to their validity have been identified.

Regarding the *internal validity*, that is, unknown factors influencing the reported results, threats can be attributed to potential bugs in the utilized implementation. Specifically, the generation, compilation, execution and comparison of the programs' outputs in the experimental environment might have influenced the results. Errors affecting the generation, compilation and executions of mutants may result in mutants that do not compile, or run with tests and thus being ignored. Errors affecting the execution and the programs' output comparison may influence the decision of whether a mutant is being killed or not. To reduce these threats, several manual checks have been undertaken on all the utilized subjects. Additionally, the output comparison was performed on the same way in both faults and mutants enabling a common basis. Furthermore, Proteum has been widely used in mutation testing experiments, thus giving confidence on the reported results.

The utilized operators give rise to other possible threats. Different implementations of these operators or using of others may result in different behaviours. However, the present study employed a wide range of mutant operators (refer to [34] for further details), which where developed for testing and independent to the present study. Another threat that can be identified is regarding the artificial test suites used. Because most of the test suites were artificially produced, they might behave differently than those produced by humans. Yet, there is no evidence supporting such a claim. However, the researchers created these tests independent to the present study, to support their experimentation [28–30]. Further, these tests have been extensively used in literature. To reduce this threat, the present study considers RQ2, where a wide range of different test cases per studied program was used.

Finally, performing fault localization on the correct and not on the faulty program may also influence the effectiveness of the approach. However, this practice was applied on all examined fault localization methods, so its effects imply on all the methods. Additionally, because it was not applied on the second benchmark set and its results are similar to the first set, it is believed that this threat is limited.

Regarding the *external validity* of the study, that is, whether the reported results generalize, a relative threat can be recognized. The empirical study consists of 11 programs and their accompanied faults; hence, it cannot be claimed that the results also apply on other programs. Different subjects, faults and types of faults, such as concurrency bugs, may have an impact on the effectiveness of Metallaxis. Considering the appropriateness of using mutant-faults, it is noted that Ali *et al.* [15] found 'no evidence to suggest that the use of mutants for this purpose is invalid'. Further studies are in need to reduce this validity threat. However, the Siemens suite form industrial programs widely used in this kind of experiments. Also, Space, Gzip, Grep and Flex programs are real-word programs showing that the proposed approach is effective.

DOI: 10.1002/stvr

Considering the *construct validity* threats of the study, that is, threats attributed to the employed evaluation metrics, can be identified. Thus, the experimental evaluation based on the 'Score' metric may not be appropriate. Parin and Orso [39] conducted a small case study investigating this issue. Their results suggest that programmers have difficulties in using the provided rankings. This metric has been employed because of its extensive use in literature and as a way to compare the examined approaches. Still, it is difficult to determine whether it is practically useful. Further studies are required to determine the appropriateness of this metric.

8. RELATED WORK

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

As it has already been described, Jones *et al.* [10] developed the Tarantula method, and Abreu *et al.* [9] introduced the Ochiai formula. Both these advances were utilized by the proposed mutation-based 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 [13], Santelices *et al.* [8] and Yu *et al.* [14]. Marsi [13] concluded that branches and definition–use pairs are more effective than statements. However, Santelices [8] showed that there is no approach that performs better in all cases, hence proposing a combination of methods. On the basis of their results, Yu *et al.* [14] proposed a different combination approach. Similarly, Wong *et al.* [7] proposed a set of coverage-based heuristics able to improve the effectiveness of Tarantula. In another approach, Abreu *et al.* [40] suggested using Bayesian reasoning in assigning suspiciousness values to the program spectra. This approach uses a probabilistic model, and thus, it goes a step forward by handling multiple faults effectively. Baah *et al.* [32] employed probabilistic analysis in combination with the structural program dependence (program dependence graph), to assist fault localization. Yoo *et al.* [31] suggest using information theory to assist the 'fault localization prioritization' problem.

Another related approach that is not using coverage information is the delta debugging method [4–6]. This method recognizes and isolates input parts responsible for failures [6], recognizes chains of program states that lead to the failure [5] and links these chains with the faulty code. Recently, Burger and Zeller [41] 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.* [27] 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.* [21] suggested a different approach to assist fault localization. Instead of using existing tests in the localization process, they propose 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 can be assisted to drive such approaches and to, hence, provide better results.

The idea of using mutants to assist the fault localization process was first introduced in the authors' prior work on fault localization [22], where it the use of mutation analysis for locating faults was advocated. The present paper expands the conducted experiments by considering four additional subjects (Flex, Grep, Gzip and Space) and a set of 700 mutant-faults. Generally, mutation analysis has been mainly used for testing purposes [17], and thus, very few approaches aim at program debugging using mutants. Such a debugging approach is attributed to Debroy and Wong [42] who suggested using mutants for automatically repairing program faults. The underlying idea of this work is to check whether a mutant of a faulty program has no failing test cases. In this case, the mutant is reported as a possible fix of the faulty program. Metallaxis-FL differs from this approach because it does not require finding a mutant that makes all the test cases pass. In fact, its purpose is not to fix bugs but to target the general case of fault localization, that is, it always suggests a possible suspiciousness ranking. Other related approaches include the work of Nica *et al.* [43]. In

DOI: 10.1002/stvr

this approach, mutants were employed in order to generate tests and thus to augment the utilized test suite. The augmented suite was then used for performing a model-based fault diagnosis [43]. This work uses mutants as a means of test case generation and not for performing the fault localization process, which is the main issue addressed by the present paper.

9. CONCLUSIONS AND FUTURE WORK

Supporting both testing and fault 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. First, *it shows that mutants can be utilized for the effective localization of known but not located faults*. Further, it validates this hypothesis in the cases of block, branch and 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 effective when used in a degraded situation, using few mutants*.

The major contributions made by the present paper can be summarized in 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 more than *80% of the utilized program faults by investigating at most 10% of the executable program code*.
- An experimental comparison of fault localization based on block, branch and mutation-based tests. Compared with 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 [24] as an alternative to mutation-based fault localization. Because weak mutation has been shown to be quite efficient [44] with respect to the test execution phase, utilizing it seems to be worthwhile. Additionally, the combination of the proposed approach with an automatic mutation-based test generation tool, such as [44–46], will be able to augment the utilized test suite and thus to assist the localization of program defects.

REFERENCES

1. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 2006; **32**:608–624. DOI: 10.1109/TSE.2006.83.
2. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005; 402–411, DOI: 10.1145/1062455.1062530.
3. Zhang X, Guptam N, Gupta R. Pruning dynamic slices with confidence. *SIGPLAN Notices* 2006; **41**:169–180. DOI: 10.1145/1133255.1134002.
4. Cleve H, Zeller A. Locating causes of program failures. *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005; 342–351, DOI: 10.1145/1062455.1062522.
5. Zeller A. Isolating cause-effect chains from computer programs. *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Charleston, South Carolina, USA, 2002; 1-10, DOI: 10.1145/587051.587053.
6. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 2002; **28**:183–200. DOI: 10.1109/32.988498.
7. Wong WE, Debroy V, Choi B. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software* 2010; **83**:188–208. DOI: 10.1016/j.jss.2009.09.037.
8. Santelices R, Jones JA, Yu Y, Harrold MJ. Lightweight fault-localization using multiple coverage types. *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, BC, 2009; 56–66, DOI: 10.1109/ICSE.2009.5070508.

9. Abreu R, Zoeteweij P, Gemund AJCv. On the accuracy of spectrum-based fault localization. *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, Windsor, 2007; 89–98, DOI: 10.1109/TAIC.PART.2007.13.
10. Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, Orlando, FL, USA, 2002; 467–477, DOI: 10.1145/581339.581397.
11. Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, USA, 2005; 273–282, DOI: 10.1145/1101908.1101949.
12. Liu C, Yan X, Fei L, Han J, Midkiff SP. SOBER: Statistical model-based bug localization. *SIGSOFT Software Engineering Notes* 2005; **30**:286–295. DOI: 10.1145/1095430.1081753.
13. Masri W. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability* 2010; **20**:121–147. DOI: 10.1002/stvr.v20:2.
14. Yu K, Lin M, Gao Q, Zhang H, Zhang X. Locating faults using multiple spectra-specific models. *Proceedings of the 2011 ACM Symposium on Applied Computing*, TaiChung, Taiwan, 2011; 1404–1410, DOI: 10.1145/1982185.1982490.
15. Ali S, Andrews JH, Dhandapani T, Wang W. Evaluating the accuracy of fault localization techniques. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, 2009; 76–87, DOI: 10.1109/ASE.2009.89.
16. Jia Y, Harman M. Higher order mutation testing. *Information and Software Technology* 2009; **51**:1379-1393. DOI: 10.1016/j.infsof.2009.04.016.
17. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 2011; **37**:649–678. DOI: 10.1109/TSE.2010.62.
18. Schuler D, Zeller A. (Un-)covering equivalent mutants. *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, Paris, 2010; 45–54, DOI: 10.1109/ICST.2010.30.
19. Yue Y, Harman M. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic & Industrial Conference*, Windsor, 2008; 94–98, DOI: 10.1109/TAIC-PART.2008.18.
20. Schuler D, Zeller A. Javalanche: Efficient mutation testing for Java. *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*, The Netherlands, 2009; 297-298, DOI: 10.1145/1595696.1595750.
21. Baudry B, Fleurey F, Traon YL. Improving test suites for efficient fault localization. *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2006; 82-91, DOI: 10.1145/1134285.1134299.
22. Papadakis M, Le Traon Y. Using mutants to locate "unknown" faults. *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, 2012; 691–700, DOI: 10.1109/ICST.2012.159.
23. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. *Computer* 1978; **11**:34–41. DOI: 10.1109/C-M.1978.218136.
24. Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. In *Mutation Testing for the New Century*. Kluwer Academic Publishers: Norwell, MA, USA, 2001; 34–44. ISBN:0-7923-7323-5.
25. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology* 1996; **5**:99-118. DOI: 10.1145/227607.227610.
26. Papadakis M, Malevris N. An empirical evaluation of the first and second order mutation testing strategies. *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, Paris, 2010; 90-99, DOI: 10.1109/ICSTW.2010.50.
27. Jeffrey D, Gupta N, Gupta R. Fault localization using value replacement. *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, 2008; 167–178, DOI: 10.1145/1390630.1390652.
28. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**:405–435. DOI: 10.1007/s10664-005-3861-2.
29. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, 1994; 191–200, DOI: 10.1109/ICSE.1994.296778.
30. Harder M, Mellen J, Ernst MD. Improving test suites via operational abstraction. *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, 2003; 60-71, DOI: 10.1109/ICSE.2003.1201188.
31. Yoo S, Harman M, Clark D. Fault localization prioritization: comparing information theoretic and coverage based approaches. *ACM Transactions on Software Engineering Methodology* 2013; **22**:1–29. DOI: 10.1145/2491509.2491513.
32. Baah GK, Podgurski A, Harrold MJ. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering* 2010; **36**:528–545. DOI: 10.1109/TSE.2009.87.
33. Maldonado JC, Delamaro ME, Fabbri SCPF, Simão AdS, Sugeta T, Vincenzi AMR, Masiero PC. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation Testing for the New Century*, Wong WE (ed.). Kluwer Academic Publishers: Norwell, MA, USA, 2001; 113–116. ISBN:0-7923-7323-5.
34. Agrawal H, DeMillo RA, Hathaway B, Hsu W, Hsu W, Krauser EW, Martin RJ, Mathur AP, Spafford E. *Design of Mutant Operators for the C Programming Language*. Purdue University: West Lafayette, Indiana, March 1989.

35. Zhang X, Gupta R. Whole execution traces and their applications. *ACM Transactions on Architecture and Code Optimization* 2005; **2**:301–334. DOI: 10.1145/1089008.1089012.
36. Lyu MR, Horgan JR, London S. A coverage analysis tool for the effectiveness of software testing. *Software Reliability Engineering, 1993. Proceedings., Fourth International Symposium on*, Denver, CO, 1993; 25–34, DOI: 10.1109/ISSRE.1993.624271.
37. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *Proceedings of the 19th International Symposium on Software Testing and Analysis*, Trento, Italy, 2010; 147–158, DOI: 10.1109/TSE.2011.93.
38. Kintis M, Papadakis M, Malevris N. Isolating first order equivalent mutants via second order mutation. *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, 2012; 701-710, DOI: 10.1109/ICST.2012.160.
39. Parnin C, Orso A. Are automated debugging techniques actually helping programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, Toronto, Ontario, Canada, 2011; 199-209, DOI: 10.1145/2001420.2001445.
40. Abreu R, Zoeteweij P, Gemund AJCv. Spectrum-based multiple fault localization. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, Auckland, 2009; 88–99, DOI: 10.1109/ASE.2009.25.
41. Burger M, Zeller A. Minimizing reproduction of software failures. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, Toronto, Ontario, Canada, 2011; 221–231, DOI: 10.1145/2001420.2001447.
42. Debroy V, Wong WE. Using mutation to automatically suggest fixes for faulty programs. *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, Paris, 2010; 65–74, DOI: 10.1109/ICST.2010.66.
43. Nica M, Nica S, Wotawa F. On the use of mutations and testing for debugging. *Software: Practice and Experience* 2013; **43**:1121–1142. DOI: 10.1002/spe.1142.
44. Papadakis M, Malevris N. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal* 2011; **19**:691–723. DOI: 10.1007/s11219-011-9142-y.
45. Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. *Information and Software Technology* 2012; **54**:915–932. DOI: 10.1016/j.infsof.2012.02.004.
46. Papadakis M, Malevris N. Automatic mutation test case generation via dynamic symbolic execution. *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*, San Jose, CA, 2010; 121–130, DOI: 10.1109/ISSRE.2010.38.