Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach

Mike Papadakis
SnT, University of Luxembourg
Luxembourg, Luxembourg
michail.papadakis@uni.lu

Yves Le Traon
SnT, University of Luxembourg
Luxembourg, Luxembourg
yves.letraon@uni.lu

ABSTRACT
When programs fail, developers face the problem of identifying the code fragments responsible for this failure. To this end, fault localization techniques try to identify suspicious program places (program statements) by observing the spectrum of the failing and passing test executions. These statements are then pointed out to assist the debugging activity. This paper considers mutation-based fault localization and suggests the use of a sufficient mutant set to locate effectively the faulty statements. Experimentation reveals that mutation-based fault localization is significantly more effective than current state-of-the-art fault localization techniques. Additionally, the results show that the proposed approach is capable of reducing the overheads of mutation analysis. In particular the number of mutants to be considered is reduced to 20% with only a limited loss on the method's effectiveness.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: Testing tools

General Terms
Verification.

Keywords
Program debugging, mutation analysis, fault localization.

1. INTRODUCTION
Software faults are the main cause of software failures. Experiencing such failures results in a great economic impact especially when it involves safety critical applications. To reduce such incidences, software developers try to test their software in order to find most of the software defects. However, testing only concerns the detection of software defects and not their correction. Therefore, when detecting a software fault, developers need to identify the faulty program parts in the application's source code and ultimately fix them.

The identification of the defective program places is usually referred to as the *fault localization* process and denotes the problem of localizing software faults given a set of tests. This constitutes a hard problem and results to be one of the most costly processes of the debugging activity. Researchers have put a great deal of effort to automate the fault localization activity and thus, to reduce its expenses. In this area, the main direction of research is to advise developers regarding the most suspicious program locations, which could have led to the experienced failures.

Many debugging approaches have been proposed and studied by the literature in order to tackle this problem. Delta Debugging [7] tries to identify the program states that lead to failures. Spectrum-based or coverage-based techniques [15, 26] collect program traces of both the passing and failing executions and then assign to program statements a suspiciousness value that represents a probability that these statements are faulty. To this end, many program entities, i.e. coverage entities, have been used. Entities such as statements [1, 15] branches [17] du-pairs [18] and possible combinations of them [26, 30] have been proposed and used in assisting fault localization. Empirical studies show that spectrum-based fault localization approaches not only help developers [3, 15] but also assist other activities such as the automated program repair [10].

Mutation analysis explores the programs' behavior by injecting artificial defects into its code. The main idea behind this approach is that by running the artificially defective programs, some valuable information can be gained. This information can be useful in evaluating the quality of the testing activity [21] or in automating various tasks such as the automated oracle creation [9] or fault localization [25]. The technique is powerful since it forces to test the programs with respect to their behavior and not with respect to code coverage [9]. Mutation analysis requires a vast number of defects to be injected and executed with actual test cases. Thus, scalability issues are raised. To deal with this problem, researchers have identified small but sufficient categories of defects that should be applied. We refer to these approaches as selective mutation.

Recently, a fault localization approach based on mutation analysis has been proposed [22]. We call this approach as the *mutation-based* or simple as the *mutation* approach. Despite the fact that mutation analysis was originally proposed for testing, mutation analysis has been shown to be helpful in various contexts and applications [21]. This is the case for fault localization where initial results [22, 25] show that the mutation-based approach can localize faults significantly better than the statement-based one. The intuition behind the mutation approach is that when failing test cases achieve to kill mutants while the passing ones leave them live; their location indicates a location responsible for the test failure. Following these lines, the present paper considers the mutation-based approach proposed by Papadakis and Le Traon [22, 25] and compares it with state-of-the-art spectrum-based fault localization methods, like [26, 30]. The results are promising since they show that the mutation-based fault localization outperforms the examined approaches.

Although effective, applying mutation analysis requires vast computational resources [19]. To deal with this problem, researchers have proposed various mutation variant techniques, like mutant sampling [23, 28] and selective mutation [19, 20]. A recent work demonstrates that the use of selective mutation can lead to practical solutions applicable to real world applications [9]. Going a step further, the present paper adapts this technique i.e. selective mutation, to the fault localization problem. To the authors' knowledge this is the first study aiming at identifying sufficient mutants in the context of fault localization. Empirical results show that the selective mutation approach proposed here is capable of reducing the number of involved mutants by 80% without loss on the fault localization accuracy. Therefore, the computational demands of the method are drastically reduced.

In summary, the present work empirically answers the following questions:

a) How mutation-based fault localization compares with state-of-the-art spectrum-based fault localization techniques?

b) Is it possible to form a selective mutation approach to support the fault localization activity?

The remainder of the paper is organized as follows: Section 2 presents some background material regarding mutation analysis and fault localization. Sections 3 and 4 respectively detail the conducted experiment and its findings. Sections 5 and 6 discuss the benefits of the proposed approach and its relation to the literature. Finally, Section 7 concludes the paper and identifies possible future directions.

2. MUTATION ANALYSIS AND FAULT LOCALIZATION

The present section introduces briefly mutation analysis and the spectrum-based approaches studied in the conducted experiment.

2.1 Mutation Analysis

Mutation works by making syntactic changes into the source code of the program under test. This practice results in producing various program versions, called *mutants*, each one containing a single syntactic difference with the original program version. The defective program versions are produced based on a set of simple syntactic rules, called *mutant operators*. The value of the approach comes from the execution of the mutant programs with test cases. By comparing the output of the mutant programs with the output of the original one, the ability of the test cases to expose defects is assessed. A mutant is called *killed* if its execution with a test case results in different output from the original. It is called *live* in the opposite case.

Generally, to kill a mutant or to trigger a fault, a test case must execute the faulty program place; it must cause an infection on the program state (at this point) and must propagate this infection to the programs' output. This last requirement referred to as propagation requirement, signifies the need to propagate the internal error state to the output of the program. It is this requirement that differentiates mutation analysis from the coverage based testing techniques. Researchers have provided evidence that mutants behave like real faults [4]. Therefore, killing mutants results in testing thoroughly the tested program.

Applying the technique requires answering the question of which mutant operators to use. Several studies try to answer this question [13]. Most of them try to construct all the possible simple syntactic changes [2]. Others try to use the experience of the researchers in defining them. The present paper considers a comprehensive set of mutant operators, defined based on all the construct elements of the C language [2]. This results in a vast number of mutants and thus, a smaller set of them is needed. To deal with this issue, the present study identifies a small but representative set of mutant operators to apply. Such an approach is usually referred to as *selective mutation* [13, 20].

2.2 Spectrum-based fault localization

Several fault localization approaches have been suggested based on the various different spectrum types. Thus, program statements [1, 15], program branches [17], du-pair [18] and possible combinations of them [26, 30] have been used. These techniques collect the dynamic coverage information of the executed test cases and try to associate it with the experienced failures. Thus, the program entities (coverage types) covered by each one of the executed tests are recorded. Depending on whether the test cases cover specific entities when they fail or when they pass, they are related to the pass or the failure of the test.

The underlying idea of these approaches is that entities covered mostly by failing tests and rarely by passed tests is more likely to be responsible for a failure than entities covered mostly by passed tests. In other words, they try to compute a value that represents the probability that a specific entity is faulty. This value is calculated for all the program entities and it is called *suspiciousness* value. Then, the programmer has to inspect the most suspicious statements in order to find the program place that is responsible for the failure. To this end, these methods produce a priority list of program statements. The list orders the statements according to their suspiciousness values, i.e. in a decreasing order from the most suspicious statement to the least one. The specific position of a statement in this list is called *rank*.

The question that it is raised here is how to calculate the suspiciousness values and which coverage entities are more appropriate to use. There are various ways to compute the suspiciousness values of a specific coverage entity. The present work uses the Ochiai formula [1] in the lines suggested by Santelices et al. [26]. The Ochiai formula is defined as:

$$Suspiciousness(e) = \frac{failed(e)}{\sqrt{totfailed(failed(e) + passed(e))}} \quad (1)$$

In this formula *totfailed* represents the number of test cases that fail, *failed*(e) represents the number of test cases that cover the code entity e and fail and *passed*(e) represents the number of test cases that cover the code entity e and pass.

By using different coverage entities (e), different approaches can be formed [26]. Furthermore, by combining the values of the different entities additional approaches can be made. Among all the possible combinations, the present study focuses on the most representative ones as found in [26] and [30]. The following subsections detail these approaches.

2.2.1 Statement-based Fault Localization

Performing fault localization using program statements, i.e. using the Ochiai formula (1) [1] with code entities e representing program statements, forms a simple and straightforward suspicious calculation. This approach is among the first and most popular ones in literature. Its choice is based on its simplicity and popularity. Additionally, most of the studies do consider it. This is the case for the previous studies using mutation analysis [22, 25]. However, more effective techniques exist. They are described in the following subsections.

2.2.2 Combination of Coverage Types, avg-SBD Fault Localization

Assigning suspiciousness values using coverage entities, the program statements, the branches and the du-pairs i.e. code entities *e* representing program statements, branches and du-pairs in (1), results in calculating three different values per program statement. By combining these values a better suspiciousness ranking can be made [26]. Following the lines suggested in [26], each program statement is assigned with the average suspiciousness value of the three different entities (statements, branches and du-pairs). We call this approach as avg-SBD. Further details regarding avg-SBD can be found in [26].

2.2.3 Combination of Coverage Types, Loupe Fault Localization

Different combinations of multiple spectra types can give several approaches to assist fault localization. Instead of combining like in the avg-SBD i.e. based on the average value of the different spectra (suspiciousness) values, an alternative would be to build different models for each type of spectra. Then, the best solution can be selected. Such an approach has been introduced in [30]. This approach first computes the suspiciousness of all program branches and all program data dependencies using the ochiai formula (1). Then the suspiciousness values of the program statements are calculated based on a) program predicates and b) on data dependencies. For each program predicate, case a), the suspiciousness values are calculated by using the absolute difference between the true and false branch. For all the program statements, case b), a suspiciousness value is computed based on the average value of all the data dependencies of the statement. The overall suspiciousness values assigned to a statement is the maximum value of the a) and b). We call this approach Loupe. Further details regarding Loupe can be found in [30].

2.2.4 Mutation-based Fault Localization

Mutation-based fault localization differs from the other approaches since it relies on mutants [22]. Instead of trying to associate the execution of program statements with a failure, it tries to associate the killing of mutants with test failures or passes. Therefore, suspiciousness values can be calculated by using the Ochiai formula (1) with code entities *e* to be mutants. Thus, in the equation (1), the *totfailed* represents the number of test cases that fail, the *failed*(*e*) represents the number of test cases that kill the mutant *e* and fail and *passed*(*e*) represents the number of test cases that kill the mutant *e* and pass. Further details regarding the mutation approach can be found in [22, 25].

3. EXPERIMENT PROCEDURE

This section details the conducted experiment. It defines the objectives of the study; it presents details regarding the selected subjects i.e. their test suites and faulty program versions and then, it introduces the tools employed by the study. Finally, a description of the performed analysis is given along with the identified threats to the validity of the experiment.

3.1 Definition of the Experiment

The present experiment investigates a) whether mutation-based fault localization is more effective at localizing faults than state-of-the-art spectrum fault localization techniques and b) to determine whether it is possible to select a small set of mutant operators to support the fault localization activity. By showing the point a), the superiority of the mutation-based fault localization is established. Similarly, with respect to point b), the method can be turned into practice by showing that a small sufficient set of mutant operators exists.

The above issues form the following two research questions:

> **RQ1:** How does the mutation-based fault localization compares with the current state-of-the-art techniques?
>
> **RQ2:** How effective are the proposed selective mutation approaches?

3.2 Subject Programs

The conduced study involves the seven programs composing the well-known Siemens suite [12]. These programs have been extensively used in fault localization studies such [7, 15, 17, 22, 30]. They are also well suited for the purposes of the present study since they are written in C and they are publicly available along with their associated faults and tests.

Several researchers have produced the associated test suites by using various black and white box techniques. Therefore, the test suites are capable to cover all program statements, all program edges and all definition-use pairs. Additional details regarding the construction of the test suites can be found in Harder et al. [11].

Table 1 records the details regarding the Siemens suite. It includes the number of lines of code, the size of the test pool, the number of mutants and the number of the examined faults per program. These programs were chosen due to a) they contain most of the language constructs used in large industrial studies [19] and b) mutation analysis can be appropriately applied on them since they are not of a very big size. Thus, all C mutant operators can be used, similar to the study of Siami Namin et al. [19]. The use of all mutant operators is vital for identifying a sufficient set of operators. This is due to the need of identifying sets with similar effectiveness with the whole set of mutants.

3.3 Utilized Tools

The work presented here involves a) fault localization using the state-of-the-art approaches, b) mutation analysis and c) fault localization using mutation analysis. We used three prototypes to accomplish these steps. For the step a) we employed the tool implementation of [30] which implements all the three examined fault localization approaches. To perform the step b) the Proteum[1] mutation analysis tool [8] was used. This is a well-known tool widely used in mutation testing experiments like [13, 19, 23]. For the step c) we developed a prototype on top of the Wet [31] framework similar to the prototype implementation used on the step a). This choice was mandatory in order to compare the approaches in a fair way. Since Wet records executable code and execution traces at machine code granularity level, this information may influence the fault localization results. Thus, we

[1] The version 2.0 of the Proteum/IM tool was used by utilizing all the unit level operators.

Table 1. Subject Programs

Subject Program	Lines of Code	Test Pool Size	Number of Mutants	Number of Faults
Schedule	296	2650	2241	9
Schedule2	263	2710	2980	10
Tcas	137	1608	2872	41
Totinfo	281	1052	6386	23
Printtokens	343	4130	4263	7
Printtokens2	355	4115	4681	10
Replace	513	5542	10928	32

used the same information on all the approaches. Additionally, the same prototype was employed in the mutation-based fault localization study [22].

3.4 Analysis Procedure

This section details the experimental procedure followed in order to answer the defined research questions.

3.4.1 Comparison Metric

Comparing two fault localization methods requires a way to quantify their effectiveness. We follow the usual procedure taken in these kinds of studies based on the "Score" metric [7, 15] This metric quantifies the effort made by programmers in order to identify faulty statements by evaluating the percentage of statements that does not need examination in order to find the faulty program location. Recall that the fault localization methods produce a priority list based on which they examine the program statements. Thus, the "score" metric is computed based on (2).

$$\text{"Score"} = \frac{total\ executed\ statements - rank}{total\ executed\ statement} \qquad (2)$$

rank specifies the order of the faulty statement in the ordered list of the fault localization. Here, it should be noted that higher "Score" values indicate less effort and thus, they are preferable.

3.4.2 Comparing Spectrum-based and Mutation-based approaches (RQ1)

The results of all the examined fault localization methods were analyzed and compared according to the "*Score*" metric. Initially, all the available test cases were executed in order to determine the passed and failed test cases. Then, all the test cases were again executed in the prototypes' environment so that all the required execution traces have been collected. Then, fault localization was performed based on these traces and produced results regarding the statement, avg-SBD and Loupe approaches. Regarding the mutation method, Proteum produced and compiled all the mutants. Then we executed all the test cases with all the mutants and record which mutants are killed by each one of the test cases. Based on these results, we performed the mutation-based fault localization.

3.4.3 Special cases

Performing the present study involves handling some special cases. Only executable statements were considered. This is a constraint imposed by the functionality of the tool. Furthermore, to reduce the mutation analysis computational needs (mutation analysis requires huge computational resources [19]) the main program version was only employed for the fault localization. A similar process is applied to the other mutation fault localization works [22, 25]. Additionally, in case of ties, i.e. statements with the same suspiciousness values, were "ranked at the upper of their ranks" [25]. This is a usual approach for this kind of experiments e.g. [5, 25, 30].

The case of omission faults (the actual fault is a statement missing from the source code) is handled by treating the next to the missing statement as being the faulty one. Faults lying in non-executable statements such as the variable initializations and constants assignments need also a special treatment. These cases were handled by treating the statements using the variables or the constants as being faulty. These are mandatory assumptions since no fault localization can pinpoint such faults. They are commonly assumed by other fault localization studies such as [5, 25, 30].

3.4.4 Finding Sufficient Mutant Set (RQ2)

A selective set of operators with respect to fault localization, investigated by RQ2, is determined based on the process of Figure 1. This process is named as the *sufficient procedure*. The procedure seeks to determine the sufficient operator set by incrementally selecting and adding to the sufficient set the less costly operators, i.e. operators producing the least number of mutants, among the most effective ones. In practice, it was observed that after some iteration, all the evolved sets were almost the same. Therefore, at each iteration of the process, the selected sets (n sets to be evolved) were forced to be the ones that differ in more than two operators among the less costly and most effective identified sets (step 10 of sufficient procedure). This restriction helped introducing some diversity into the selected sets and reducing the risk of over-fitting to the employed set of faults.

To prevent over-fitting to these faults it is needed to provide the procedure a more representative range of faults. Hence, the utilized fault set was augmented with 100 additional faults per studied program. These were selected at random from the produced mutant sets and are called the mutant-faults. To this end, the best sufficient sets on the last 3 iterations of the sufficient procedure were selected. These sets were then improved according to the mutant-fault sets. The sufficient set reported in the next section was produced by employing the procedure of Figure 1 with parameters N = 60, n = 6. The process was repeated four times with a = 0 for the faults and a = 5, 4, 1 and 1 for the mutants resulting in four different selective sets. These four sets consume different percentages of mutants and achieve different levels of effectiveness. We call these sets as the *Selective1, Selective2, Selective3* and *Selective4* mutant sets.

Finally, to examine the localization ability of the selective mutants, in addition to the employed faults and mutant-faults, a different set of mutant-faults was also used. This set was selected at random and it was composed of 700 mutants (100 per subject program). The use of this set is important since it provides an independent evaluation set to the one used by the sufficient procedure. In the rest of the paper the first set of mutant-faults is denoted as the *mutant-fault-set1* or simple *mutant-fault-set*, while the second one, as the *mutant-fault-set2*.

3.5 Threats to Validity

One issue related to the validity of the experiment is due to the utilized test suites. It is possible that these tests are not representative of those used by actual testers. Their choice was mainly due to their extensive use in experimental studies. Additionally, these tests were independently built by software testing researchers [11]. Another threat that can be identified is the use of mutants as substitutes of faults. This is something that has already been studied in literature [6, 25]. The validity of this

```
            OP: set of mutant operators
            Set CurrSet = [ ];
            Set SuffOp = [ ];

Step 1.     Score = 0;
Step 2.     for each mutant operator op in OP {
Step 3.         CurrSet = op;
Step 4.         SuffOp.add( CurrSet );
            }
Step 5.     for each set CurrSet in SuffOp {
Step 6.         for each mutant operator op in OP {
Step 7.         Perform fault localization with respect to op + CurrSet;
Step 8.         Evaluate fault localization Scores;
                }
            }
Step 9.     CurrSet = Select the N sets with the highest Scores;
Step 10.    SuffOp = Select n sets with the less number of
mutants from CurrSet;
Step 11.    CurrScore = Sum of scores in SuffOp;
Step 12.    if ( |CurrScore - Score| < a ){
Step 13.        Score = CurrScore;
Step 14.        Goto step 2;
            }
Step 15.    return CurrSet;
```

Figure 1. Sufficient Procedure: determining sufficient set of mutant operators

practice has also been researched by Ali et al. [3] who conclude that "no evidence to suggest that the use of mutants for this purpose is invalid". Additionally, the use of mutant-faults may introduce a bias with the mutation fault localization due to the use of these mutants both as faults and as location indicators. To reduce this threat, all the mutant-faults were excluded from the mutant set of the fault localization method.

Another possible threat is due to the employed tools. In particular, bugs may influence the generation, compilation, executions and determination of the killed mutants and thus, affect the reported results. Manual checks were made in order to lighten this threat. Additionally, all the tools used here have been used in several mutation testing and fault localization experiments, like [19, 25, 30]. Another issue can be related to the employed mutant operator sets. Other sets may behave differently. However, the utilized set was proposed independently of the present study and it is composed of a wide range of operators involving all the C language constructs [2].

Other issues are related to the generalization of the reported results and to the use of the "Score" metric. The experiment involves 7 programs with their accompanied faults and thus, it is difficult to claim that the results are generalizable. Similarly, it is difficult to claim that the employed metric represents the intended effectiveness measure. However, in literature, the selected benchmarks and the "Score" metric form the standard way of evaluating the effectiveness of the fault localization approaches. Clearly, additional studies are in need to answer the abovementioned concerns.

4. RESULTS AND ANALYSIS

The present section presents the results regarding the comparison of the fault localization techniques, Section 4.1, and the identification of a sufficient mutant set, Section 4.2.

4.1 Comparison with other methods – (RQ1)

The comparative results, i.e. "score" values, of the examined fault localization approaches are presented in the graph of Figure 2. Following the lines of [7, 26, 30], the graph presents the respective results grouped according to: *99-100%, 90-100%, 80-100%, 70-100%, 60-100%, 50-100%, 40-100%, 30-100%, 20-100%, 10-100%, 0-100%, "score" ranges*. Specifically, the y-axis of Figure 2 records the ratio of the faults that are effectively localized in the "score" ranges recorded by the x-axis. In other words x-axis represents the ratio of statements, over the whole executable ones, that do not need to be analyzed during the fault localization process. Hence, higher values on this plot indicate less effort by the programmer and thus, higher fault localization effectiveness.

For example, based on the results of Figure 2, a developer will be able to locate approximately *90%* of faults when he examines the 10% of the programs' code[2] and employs the mutation-based approach. Similarly, with the same effort he will locate 58% of faults if he employs Loupe while with avg-SBD he will locate 54% and only *44%* with the statement one. Overall, these results indicate that the mutation-based method is far more effective than all the other examined approaches. It can be argued that the differences are practically significant since the mutation method localizes effectively more faults in all the examined ranges. Additionally, the difference in the whole range from *70% to 100%* is higher than 15% in favor of the mutation approach. The mean value for all the faults of the mutation approach is *95%* while for Loupe is 84%, 83% for avg-SBD and *77%* for the statement one.

4.2 Selective Mutation evaluation – (RQ2)

The resulting sets of mutants produced by applying the Sufficient Procedure are recorded in Table 2. The identified operator sets are denoted as Selective1, Selective2, Selective3 and Selective4 and consume the 22%, 27% 35% and 45% mutants of the whole mutant set. Four selective sets are reported mainly due to the different number of mutants they consume and the effectiveness levels that they achieve. Tables 3 and 4 respectively record the ratio of the effectively localized faults and mutant-faults at various considered "score" ranges regarding the whole and the selective sets of mutants. These results strongly suggest that the identified mutant sets are approximately of the equal effectiveness as the one containing all mutants.

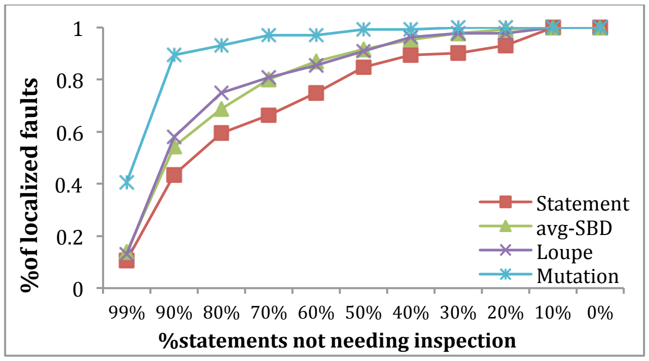

Figure 2. Comparison of the Mutation, avg-SBD, Loupe and Statement fault localization methods.

[2] Only the executable statements are considered.

Table 2. Selective mutant Sets

Selective Set	Operators	%Mutants considered
Selective1	*u-Cccr, u-OAAN, u-OCNG, u-Oido, u-OLLN, u-ORSN, u-SSDL, u-STRP, u-VGSR*	22%
Selective2	*u-Cccr, u-OARN, u-OASN, u-OCNG, u-Oido, u-OLLN, u-ORRN, u-ORSN, u-SBRC, u-SSDL, u-STRP, u-VTWD*	27%
Selective3	*u-Cccr, u-OABA, u-OARN, u-OASN, u-OCNG, u-OEAA, u-Oido, u-OLLN, u-ORRN, u-ORSN, u-SBRC, u-SSDL, u-STRP, u-VGSR, u-VTWD*	35%
Selective4	*u-Cccr, u-OABA, u-OARN, u-OASN, u-OCNG, u-OEAA, u-Oido, u-OLLN, u-ORRN, u-ORSN, u-SBRC, u-SSDL, u-STRP, u-VLSR, u-VTWD*	45%

This is somehow expected, since the selective set was chosen based on these mutant-faults. The question that is raised here, is whether the selective sets are representative of the whole set. To address this issue, statistical analysis was performed in order to determine whether the selective sets have statistical differences with the whole set of mutants. This analysis was performed on all the fault sets (faults, mutant-fault-set1 and mutant-fault-set2). Recall that mutant-fault-set2 was chosen from the whole mutant set by randomly selecting 100 different to the mutant-fault-set1 mutants per program. Table 5 records the p-values of the statistical comparisons of the selective sets and the whole set of mutants. These results reveal that there is no statistically significant difference between all mutants and the selective sets.

5. DISCUSSION

Mutation analysis is a powerful technique with application on many software engineering problems. However, its main obstacle is the vast number of mutants that it introduces. Thus, scalability issues can be raised. This is an open research issue of the method and requires additional research. However, the results presented in this paper do make a major step towards this direction by identifying sufficient subsets of mutant operators. Testing large programs with the use of mutation analysis is only possible with the use of small mutant subsets. This is evident from the recent studies e.g. [4, 9] of mutation which all use a selective form of mutation testing. Thus, it can be argued that without a selective mutation approach large programs will remain intractable.

Software testing activities are accomplished before those of debugging. This fact gives debugging the opportunity to reuse information from the testing process. Our approach can gain many benefits from this observation since it can reuse some, if not all, the information required by mutant execution. Additionally, the process can be combined with other testing approaches like the higher order mutation [14], the equivalent mutant isolation [16] and the automated generation of assertions [9]. If such approaches are used, our approach will need to perform only the mutant executions that were not made during testing. These executions involve the test execution of mutants that did not employed in the testing stage or those that were ignored by the testing process for optimization reasons.

6. RELATED WORK

Both fault localization and mutation analysis are topics well studied by the literature. However, only a few and recent works combine them. This section gives first a brief description of fault localization works and then of mutation analysis ones.

Table 3. Percentage of Located Faults W.R.T Score Ranges

Score	Mutation	Selective1	Selective2	Selective3	Selective4
Average	***95.42%***	***95.53%***	***96.08%***	***96.06%***	***96.12%***
99-100%	40.46%	39.69%	43.51%	45.04%	45.04%
90-99%	89.31%	89.31%	90.84%	90.84%	90.84%
80-90%	93.13%	94.66%	93.13%	94.66%	93.89%
70-80%	96.95%	96.95%	97.71%	96.95%	96.18%
60-70%	96.95%	99.24%	99.24%	99.24%	99.24%
50-60%	99.24%	99.24%	99.24%	99.24%	99.24%
40-50%	99.24%	99.24%	99.24%	99.24%	99.24%
30-40%	100.00%	100.00%	100.00%	100.00%	100.00%
20-30%	100.00%	100.00%	100.00%	100.00%	100.00%
10-20%	100.00%	100.00%	100.00%	100.00%	100.00%
0-10%	100.00%	100.00%	100.00%	100.00%	100.00%

Table 4. Percentage of Located Mutant-Faults W.R.T Score Ranges

Score	Mutation	Selective1	Selective2	Selective3	Selective4
Average	***95.26%***	***94.01%***	***94.17%***	***94.82%***	***94.97%***
99-100%	30.14%	28.00%	26.71%	28.86%	29.29%
90-99%	87.86%	84.57%	85.43%	86.00%	87.14%
80-90%	94.43%	90.29%	91.71%	92.43%	93.43%
70-80%	96.00%	96.71%	96.43%	98.00%	98.00%
60-70%	99.43%	98.00%	97.86%	98.86%	98.86%
50-60%	99.86%	99.14%	98.71%	99.29%	99.29%
40-50%	100.00%	99.71%	99.57%	99.71%	99.71%
30-40%	100.00%	99.86%	99.71%	99.86%	99.86%
20-30%	100.00%	100.00%	100.00%	100.00%	100.00%
10-20%	100.00%	100.00%	100.00%	100.00%	100.00%
0-10%	100.00%	100.00%	100.00%	100.00%	100.00%

Table 5. Statistical Comparison (P-Values) of ALL operators and Selective Sets

	Selective1	Selective2	Selective3	Selective4
Faults	0.5908	0.4702	0.4658	0.4330
Mutant-Faults-Set1	0.0837	0.0968	0.4839	0.5602
Mutant-Faults-Set2	0.0697	0.1283	0.7961	0.8213

6.1.1 *Fault Localization*

One of the first and most popular fault localization approach is Tarantula, which was introduced by Jones et al. [15]. Tarantula is a statement-based method that uses a similar coefficient to Ochiai formula. However, Abreu et al. [7] showed that the Ochiai formula is more effective. The use of program branches and du-pairs instead of program statements was suggested by Marsi [18], who showed that they are more effective. This approach was later extended by Santelices et al. [6] who presented a unified way of applying all these advances. Additionally, the study of Santelices et al. [6] revealed that there is not a specific spectra entity that provides always the best results. Therefore, they propose to combine the measures in order to increase the fault localization accuracy. This approach was later inspired Yu et al. [12] who suggested a different way to combine the methods and get better results. In the same lines, Wong et al. [5] provided some heuristics that were shown to be better than Tarantula.

Other related techniques, like the one of Baah et al. [5], build probabilistic models based on program dependencies to assign the statement suspiciousness values. Yoo et al. [29] employed information theory in order to prioritize the execution of test cases. Their aim is to execute the tests of a regression test suite in a way that it maximizes the accuracy of fault localization. Jeffrey et al. [25] proposed an approach that replaces program values. Thus, the variables of the program statements are replaced with others during runtime and comparing the effect on the output of the program. If the output is corrected then the faulty statement is reported as the most suspicious one.

Fault localization is dependent on the utilized test suites. Therefore, one way to assist the fault localization process is by adding and/or removing test cases. Baudry et al. [6] suggested optimizing the test suite generation to improve the fault localization accuracy. This approach is complimentary to ours since it aims at producing test cases that will be used by the approach in the localization process.

6.1.2 *Mutation Analysis*

The use of mutation analysis in directing the testing process has been suggested over three decades [21]. Since its initial suggestion, many works have been introduced with an increasing trend over the last years, as it is revealed by the recent survey of Jia and Harman [13]. Most relevant are those that focus on reducing the cost of the approach by either a) reducing the required time to generate and execute the sought mutants or b) by reducing their number.

Regarding the issue a), the mutant schemata technique [13] has been suggested as a way to reduce the cost of compiling the mutant programs. Similarly, weak-firm mutation [13, 24] has been proposed as a way of reducing the computational cost of mutant execution. These, approaches are orthogonal to the one presented here. Since they aim at reducing the time required to introduce and execute the mutants, they can be applied independently to the selected operators.

With respect to the number of mutants, issue b), several approaches have been studied. The most naïve one is random sampling [23, 28]. By randomly selecting a set of mutants their number can be reduced with a small effect on their effectiveness [23, 28]. Other approaches aim at selecting a small but representative set of operators. Wong and Mathur [28] proposed the use of two operators for Fortran programs. This work was later refined by Offutt et al. [20] who suggested the use of five operators. These five operators are the most commonly used in literature studies like [4, 9, 24]. In the same lines Barbosa et al. [27] suggested the use of ten mutant operators for the C language. Later, the study of Siami Namin et al. [19] suggested the use of 28 mutant operators. All these works are different from the present one since they target on testing and not on fault localization.

A different approach to reduce the number of mutants has also been suggested based on the notion of higher order mutation [14]. These methods try to reduce the mutants by injecting more than one defect at the same time. Thus, instead of having mutants based on one simple syntactic changes, first order mutants, the mutants may have multiple syntactic changes, higher order mutants. These methods actually increase the number of the mutants since higher order mutants are all the possible combinations of the first order mutants. Thus, a selection process has to be performed. One way is the random sampling or based on some special characteristics of the mutants like their program location [23]. Other possible ways are based on the use of search-based approaches [14]. All these approaches are different from the present one since they aim at testing and not at fault localization.

7. CONCLUSIONS AND FUTURE WORK

This paper addresses the issue of mutant selection for mutation-based fault localization. This is an important problem of mutation analysis [19, 21]. Without a proper mutant selection, vast computational demands are needed. Additionally, the effective application of the method is highly dependent on the quality of the employed mutants. To this end, the present paper identifies representative sets of mutant operators for the context of fault localization. The identified operator sets are capable of having almost the same effectiveness with the whole set of mutant operators despite requiring 80% less mutants.

Generally, the use of mutation analysis for debugging purposes is a relatively new direction of research. It has also a great potential since it can be combined with the testing process [22, 25]. The results of the present paper complement the previous research on this topic by drastically reducing the cost of the approach. They also show that the mutation fault localization can be significantly more effective than some of the most advanced approaches found in literature. Putting these two findings together opens the way towards the practical application of the method.

Future work includes performing empirical studies on the Object Oriented programming paradigm. Additionally, we seek to compare our findings with other subjects and link them with the testing process. Such an attempt will identify sufficient mutant sets capable to effectively drive both the testing and debugging activities. Finally, the use of higher order mutation is another direction of research. Research on higher order mutation [14] shows that it can be beneficial. However, it is not clear how to integrate this method in order to effectively localize faults.

8. REFERENCES

[1] Abreu, R., Zoeteweij, P. and Gemund, A.J.C. van 2007. On the Accuracy of Spectrum-based Fault Localization. *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. IEEE Computer Society.

[2] Agrawal, H., DeMillo, R.A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A.P. and Spafford,

E. 1989. *Design of Mutant Operators for the C Programming Language*. Purdue University.

[3] Ali, S., Andrews, J.H., Dhandapani, T. and Wang, W. 2009. Evaluating the Accuracy of Fault Localization Techniques. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society.

[4] Andrews, J.H., Briand, L.C., Labiche, Y. and Namin, A.S. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Trans. Softw. Eng.* 32, 8 (2006), 608–624.

[5] Baah, G.K., Podgurski, A. and Harrold, M.J. 2008. The probabilistic program dependence graph and its application to fault diagnosis. *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM.

[6] Baudry, B., Fleurey, F. and Traon, Y. Le 2006. Improving test suites for efficient fault localization. *Proceedings of the 28th international conference on Software engineering*. ACM.

[7] Cleve, H. and Zeller, A. 2005. Locating causes of program failures. *Proceedings of the 27th international conference on Software engineering*. ACM.

[8] Delamaro, M. and Maldonado, J.C. 1996. Proteum - A Tool for the Assessment of Test Adequacy for C Programs. *Proceedings of the Conference on Performability in Computing Systems* (1996), 79–95.

[9] Fraser, G. and Zeller, A. 2011. Mutation-Driven Generation of Unit Tests and Oracles. *Software Engineering, IEEE Transactions on*. PP, 99 (2011), 1.

[10] Le Goues, C., Dewey-Vogt, M., Forrest, S. and Weimer, W. 2012. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), 3–13.

[11] Harder, M., Mellen, J. and Ernst, M.D. 2003. Improving test suites via operational abstraction. *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society.

[12] Hutchins, M., Foster, H., Goradia, T. and Ostrand, T. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press.

[13] Jia, Y. and Harman, M. 2010. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*. 99, PrePrints (2010).

[14] Jia, Y. and Harman, M. 2009. Higher Order Mutation Testing. *Inf. Softw. Technol.* 51, 10 (2009), 1379–1393.

[15] Jones, J.A. and Harrold, M.J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM.

[16] Kintis, M., Papadakis, M. and Malevris, N. 2012. Isolating First Order Equivalent Mutants via Second Order Mutation. *Software Testing, Verification, and Validation, International Conference on*. 0, (2012), 701–710.

[17] Liu, C., Yan, X., Fei, L., Han, J. and Midkiff, S.P. 2005. SOBER: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*. 30, 5 (2005), 286–295.

[18] Masri, W. 2010. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*. 20, 2 (2010), 121–147.

[19] Namin, A.S., Andrews, J.H. and Murdoch, D.J. 2008. Sufficient mutation operators for measuring test effectiveness. *Proceedings of the 30th international conference on Software engineering*. ACM.

[20] Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H. and Zapf, C. 1996. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118.

[21] Offutt, J. 2011. A mutation carol: Past, present and future. *Information and Software Technology*. 53, 10 (2011), 1098–1107.

[22] Papadakis, M. and Le-Traon, Y. 2012. Using Mutants to Locate "Unknown" Faults. *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*.

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

[24] Papadakis, M. and Malevris, N. 2011. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal*. 19, 4 (2011), 691–723.

[25] Papadakis, M. and Le Traon, Y. 2013. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability*. (2013), n/a–n/a.

[26] Santelices, R., Jones, J.A., Yu, Y. and Harrold, M.J. 2009. Lightweight fault-localization using multiple coverage types. *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society.

[27] Vincenzi, A.M.R., Maldonado, J.C., Barbosa, E.F. and Delamaro, M.E. 2001. Unit and integration testing strategies for C programs using mutation. *Software Testing, Verification and Reliability*. 11, 4 (2001), 249–268.

[28] Wong, W.E. and Mathur, A.P. 1995. Reducing the cost of mutation testing: an empirical study. *J. Syst. Softw.* 31, 3 (1995), 185–196.

[29] Yoo, S., Harman, M. and Clark, D. 2013. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *{ACM} {T}ransactions on {S}oftware {E}ngineering {M}ethodology*. 22, 3 (Jul. 2013), 19:1–19:29.

[30] Yu, K., Lin, M., Gao, Q., Zhang, H. and Zhang, X. 2011. Locating faults using multiple spectra-specific models. *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM.

[31] Zhang, X. and Gupta, R. 2005. Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.* 2, 3 (2005), 301–334.