2010 Third International Conference on Software Testing, Verification and Validation

Using Mutation to Automatically Suggest Fixes for Faulty Programs

Vidroha Debroy and W. Eric Wong
Department of Computer Science
The University of Texas at Dallas, USA
{vxd024000,ewong}@utdallas.edu

ABSTRACT
This paper proposes a strategy for automatically fixing faults in a program by combining the processes of mutation and fault localization. Statements that are ranked in order of their suspiciousness of containing faults can then be mutated in the same order to produce possible fixes for the faulty program. The proposed strategy is evaluated against the seven benchmark programs of the Siemens suite and the Ant program. Results indicate that the strategy is effective at automatically suggesting fixes for faults without any human intervention.

Keywords: program debugging, mutation, fault localization, fault-fixing, software testing.

1. Introduction

Debugging, when a program failure is observed, is essentially a two-step process that consists of first determining the location and nature of a suspected fault, and then fixing the fault itself [20]. The first step, termed as 'fault localization', has been reported to be one of the most expensive program debugging activities [29]. Recognizing this, several techniques have been proposed that aim to reduce the manual effort that is spent by programmers to locate faults. In general, fault localization is a technique that aims to help software developers find faults by analyzing abstractions of program execution traces collected at runtime to create a ranking of most probable faulty components (e.g., program statements or predicates). These components may then be examined, by programmers, in order of their suspiciousness (of containing faults) until a fault is found [1,8,17,18,19,24,31,35,36,38]. One major critique of this approach is the assumption of 'perfect fault detection,' i.e., by examining a statement or a predicate, a programmer can tell whether it is faulty or not. Another issue is that even if the fault has been detected, the burden of fixing the fault is still left solely to the programmers. In this paper, we focus on the second issue.

A common problem faced by fault localization studies is a lack of suitable data sets on which to evaluate the effectiveness of their techniques. It is often the case that even though subject programs are readily available, there are not enough faulty versions of those programs available to conduct a comprehensive analysis. Recently researchers (not just in the area of fault localization) have relied on the *mutation* technique to address this issue, where each mutant of a correct program represents a faulty version suitable for study [10,19,21,22]. The rationale behind this approach is that mutants so generated can represent 'realistic faults' and when used in experiments yield trustworthy results [2,10,19,21,22]. However, this leads us to a very intriguing question: If mutating a correct program can result in a realistic fault, can mutating a faulty program result in a realistic fix for some faults?

Even if the above were possible, mutant generation and execution can be very expensive and the number of possible mutants for a program can be quite explosive. Are we to generate every single mutant for the entire program? Where do we start and how do we know when we have fixed the fault? Such problems present quite a daunting obstacle to using mutation as a fault-fixing aid. But perhaps some useful direction can be provided by combining this concept with that of fault localization.

If a fault localizer has already ranked program components such as statements[1] in order of how likely they are to contain faults, these statements may be mutated in the same order until a possible fix (one of the mutants) is found. Also, under the assumptions that the localizer is of sufficient quality and ranks a faulty statement as highly suspicious, and the fault is fixable using the mutant operators[2] applied; starting from the most suspicious statement we would only need to generate and execute a fraction of the total number of possible mutants, thereby reducing the overhead in fault-fixing significantly. Furthermore, this can be done automatically without the need for human intervention. To investigate this idea, we perform a case-study on the seven benchmark programs of the Siemens suite [28] and the Ant program [13] using the Tarantula fault localizer [17] and present our results.

The major contributions that are made by this paper are listed as follows:

1. We propose the use of the mutation technique to automatically produce possible fixes for faulty programs.
2. We propose the combination of the mutation technique with fault localization techniques to reduce the amount of overhead involved.
3. We present an empirical evaluation of the proposed strategy and in doing so we also propose an initial set of mutant operators that can be used for fault-fixing, as well as possible criteria to prevent the generation and execution of excessive mutants.

The remainder of this paper is organized as follows: Section 2 presents some background information that can help readers better understand this paper. Section 3 reports the case studies undertaken to evaluate the proposed strategy, while Section 4 discusses the strategy and some of its critical aspects in depth. Subsequently, Section 5 gives an overview of some related studies, and finally our conclusions and future work appear in Section 6.

[1] The same applies to other program components such as predicates.

[2] The term "mutant operators" is also referred to as "mutation operators" in other published literature.

978-0-7695-3990-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICST.2010.66

2. Preliminaries

In order to better understand the work presented in this paper, we first provide some background knowledge regarding the Tarantula fault localizer and the mutation technique. We also discuss how mutation can be applied to fix faults and how the overhead can be reduced significantly by incorporating fault localization into the strategy.

2.1. Tarantula

The Tarantula fault localizer [17] makes use of dynamic information that is collected as the result of a test execution against the program under test. More precisely, it utilizes the success/fail information about each test case execution (i.e., the execution result) and the statements that are executed by each test case. A test case is considered to be successful if the program behavior, when executed against the test case, is as expected. Correspondingly, if the observed program behavior on a test case deviates from the expected behavior, then the test case is a failed test case. A suspiciousness value is assigned to each executable statement[3] *s* according to the formula

$$suspiciousness(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{successful(s)}{totalsuccessful} + \frac{failed(s)}{totalfailed}}$$

Where *successful*(s) is the number of successful test cases, and *failed*(s) is the number of failed test cases that executed statement *s* one or more times; *totalsuccessful* is the total number of test cases that succeed, and *totalfailed* is the total number of test cases that fail. Once the suspiciousness value for each statement is computed, all the statements can then be sorted in descending order from most suspicious to least suspicious to produce a ranked list that can be examined in order by programmers for locating faults.

The effectiveness of Tarantula (and other similar fault localizers) can be measured, with respect to a fault, in terms of the percentage of statements that have to be examined until the first statement containing a fault is reached. With respect to the Siemens suite, the authors of [17] report that Tarantula consistently outperforms (in terms of examining less code) other fault localization techniques such as set union, set intersection, nearest neighbor [24] and cause transitions [8]. It is important to point out however, that the suspiciousness value assigned to each statement by Tarantula may not necessarily be unique. It is quite possible for two or more statements to have the same assigned suspiciousness and therefore, share the same rank in the list. In such an event, we have two levels of effectiveness – the *best* case and the *worst* case. In the best case we assume that we examine the fault containing statement before any other statements with the same suspiciousness. In the worst case the fault containing statement is examined last and we may have to examine many correct statements with the same suspiciousness before we find the fault. A discussion on why Tarantula was used (instead of other fault localizers) is presented in Section 4.2.

[3] Executable statements of a program exclude all the comments, blank lines, and declarative statements such as function and variable declarations. Hereafter, the use of the word *statement* refers to an *executable statement*, unless specified otherwise.

2.2. The Mutation Technique

Mutation is a technique that assesses the fault detection effectiveness of a test set for a particular program, by introducing syntactic code changes into a program, and then observing if the test set is capable of detecting these changes [5,9,10,34].

First, a mutant operator (say *m*) is applied to a program *P* to generate a mutant *P'*. If *P'* is different from *P* by exactly one change, then *P'* is a first-order mutant. Otherwise, *P'* is a higher-order mutant if there are at least two changes between *P* and *P'*. In this paper, we only use first-order mutants. It is likely that the application of *m* to *P* shall not just lead to the generation of one such *P'*, but rather several similar yet distinct mutants. If the program consists of more than one location where *m* may be applied, then *m* is applied one by one to each location, producing a distinct mutant each time.

Given a test set *T*, each mutant *P'* can be executed against every test case in *T*. If there exists a test case (say *t*) in *T* such that $P(t) \neq P'(t)$ (i.e., the output of *P'* on test case *t* is different from that of *P*), then *P'* is said to have been *killed* by *t*. In other words, the fault in *P'* has been discovered by *t* as *P'* fails to produce the expected output. This is consistent with the taxonomy in [4] where a failure is defined as an event that occurs when a delivered service deviates from the correct service. A mutant that is not killed by any of the test cases in *T* is said to be *live*, i.e., the fault in this mutant is not discovered.

It is important to note that a mutant *P'* may be functionally equivalent to the original program *P* and therefore, no test case can kill it. Such mutants are called *equivalent* mutants. Thus, a live mutant may not be killed by any test case in a test set *T* either because it is an equivalent mutant or because the test set *T* is insufficient to kill the mutant. A mutation score can be assigned to a test set which is the percentage of *non-equivalent* mutants killed by test cases in this set. A test set is said to be *mutation adequate* if its mutations score is 100% [23].

2.3. The Use of Mutation to Fix Faults

Consider the case where a program *P* (the desired correct program) is to be produced and we have a test set *T* to help verify whether the program that we have implemented runs correctly on every test case in *T*. Unfortunately, the execution of the program fails on one or more of the test cases in *T* implying that the program that we have in hand is not correct. Instead of the desired program *P*, what we really have is a faulty version of *P*, say *P'*. The application of a mutant operator to this faulty program *P'* will result in a mutant, say *P"*. The fault in *P'* will be fixed if *P"* = *P*. Now, in the current context of using mutation to automatically fix faults, we must slightly modify the traditional way in which mutation is applied.

Recall that in the source code-based mutation testing, a mutant of a program is said to have been killed if for any test case in the available test set, the output of the mutant differs from that of the original un-mutated version of the program. Following the same approach, when mutation is applied to an already faulty program *P'*, any resultant mutant *P"* can be killed if the output of any test case in the available test set differs across *P'* and *P"*. However, if our goal is to fix the fault in *P'*, then our focus is not on the relationship between *P"* and *P'*, but rather on the relationship between *P"* and *P* (the desired correct

program). We now redefine the requirement for killing a mutant P'' as

> A mutant P'' is killed if for any test case in the available test set, the output of that test case on P'' differs from the expected output on the correct program P.

Any such mutant P'' that has been killed (according to the above new definition) cannot represent a possible fix to the fault in P' as it still results in test case failure. This is because under normal circumstances, if the execution environment is correctly set up, then there is no reason for a correct program to result in test case failure and therefore, any program that results in failure can be assumed to be incorrect. Such mutants (that result in test case failure) possibly just add to the number of faults in the program without addressing the original fault(s) that we are trying to fix. They are therefore eliminated from the list of possible fixes.

On the other hand, if mutating an already faulty program results in a mutant that is un-killable or live with respect to all test cases in the available test set, then such a mutant might represent a fix to the fault, i.e., a possibly correct version of the program. We use words such as 'might represent' and 'possibly correct' because it is important to note the fact that a mutant is live, does not always imply that the mutant is the correct version of the program. It is possible that such a live mutant still contains a fault or faults and merely seems to be the correct version of the program because none of the available test cases can distinguish its behavior from that of the correct program. Let us use the following example to better explain this phenomenon.

Consider the case where we wish to write a program that takes as input two numbers and outputs their product. However, erroneously the multiplicative operator has been replaced with the additive operator (i.e., we are erroneously computing the sum of the two numbers instead of their product). We also have a test set comprised of two test cases, i.e., for two test inputs we have the expected outputs. Unfortunately, both test cases result in failure and therefore, we know that the implemented program differs from the desired program and is thus, faulty. This scenario is depicted in the columns corresponding to P and P' in Figure 1.

Since we know the program P' is faulty, we are now ready to see how mutation can help us fix the fault in P'. For discussion purposes, let us currently restrict ourselves only to the use of the simple arithmetic mutant operators, namely +, −, *, and /. In the faulty program P' we find that the third statement is the only location where such mutant operators can be applied and their application to P' yields three mutants (P_1'', P_2'' and P_3''). Based on Figure 1 we observe that 2 of these mutants (P_1'' and P_3'') produce the correct outputs on both test cases in our test set.

Only one of the mutants (P_1'') is really correct. But our current test set suggests that both P_1'' and P_3'' are the correct version of the program as every test case in our set executes them without failure. Such a problem might be avoided by augmenting the test set. However, a discussion on how an adequate test set might be constructed is beyond the scope of this paper. The objective of this example is to illustrate how a live mutant may not necessarily be the correct program. With respect to our new requirement for killing a mutant we now assert the following:

> In order for a mutant to be the correct version, its liveness is a *necessary*, but not a *sufficient*, condition.

	Desired Program (P)	Faulty Program (P')	Mutant (P_1'')	Mutant (P_2'')	Mutant (P_3'')

	read(a);
	read(b);
	y= a * b;
	print(y);
		read(a);
		read(b);
		y=a + b;
		print(y);
			read(a);
			read(b);
			y=a * b;
			print(y);
				read(a);
				read(b);
				y=a - b;
				print(y);
					read(a);
					read(b);
					y=a / b;
					print(y);

Test t_1 Input: a = 0 b = 1	output: 0	output: 1	output: 0	output: -1	output: 0
		Fail	*Succeed*	*Fail*	*Succeed*
Test t_2 Input: a = 1 b = 1	output: 1	output: 2	output: 1	output: 0	output: 1
		Fail	*Succeed*	*Fail*	*Succeed*

Figure 1: The correct and faulty programs, mutants generated from the faulty program, as well as their corresponding execution results

2.4. Overhead Reduction Using a Fault Localizer

The example in the Section 2.3 illustrates how mutation may be used as an effective technique to fix faults. However, one potential problem with using mutation alone is that there may be a large number of locations where a mutant operator may be applied. Without any guidance as to where to start, we may have to examine[4] a large number of mutants in order to find a potential fix. We now present another example to illustrate how the mutation technique might be combined with a fault localizer to effectively and efficiently localize and suggest fixes for faults. Let us take an implemented program P' which differs from the intended target program P such that of three test cases in a test set T, one of them results in a failure. Refer to Figure 2 for details. The faulty statement in this example is statement 5 and the failed test case is t_3.

			Coverage			Suspiciousness based on Tarantula
Stmt. #.	Desired Program (P)	Faulty Program (P')	t_1	t_2	t_3	
1	read(a);	read(a);	•	•	•	0.5
2	read(b);	read(b);	•	•	•	0.5
3	sum= a +b;	sum= a +b;	•	•	•	0.5
4	if(sum < 100)	if(sum < 100)	•	•	•	0.5
5	sum = sum *2;	sum = sum / 2;	•		•	0.66
6	print(sum);	print(sum);	•	•	•	0.5
Test t_1 Input:a = 0 b = 0	output: 0	output: 0				
		Succeed				
Test t_2 Input:a = 50 b = 50	output: 100	output: 100				
		Succeed				
Test t_3 Input:a = 10 b = 10	output: 40	output: 10				
		Fail				

Figure 2: An example to illustrate the proposed fault-fixing strategy

The statement coverage information for each test case is also provided: a black dot in the row corresponding to a statement implies that the statement is covered by the test case, and the absence of a black dot means that the statement is not covered by that test case (for example, statement 5 is covered by t_1 and t_3, but not t_2). Based on the coverage information and the execution results (success or failure), the suspiciousness computed according to the Tarantula formula (see Section 2.1) is listed in the last column. Thus, the statement ranking (based on the suspiciousness) would have statement 5 at the top, followed by

[4] In this paper, by examining a mutant, we mean the generation of the mutant and then the execution of the mutant against all of the test cases in our test set to determine if the mutant is a potential fix or not.

the other statements which are tied for the same ranking (as they have the same suspiciousness).

Considering the same class of mutant operators as in the previous example (i.e., the simple arithmetic mutant operators: +, −, ∗ , and /), we find that the application of these operators to the faulty program *P'* leads to the generation of 6 mutants. This is because *P'* has two locations (statements 3 and 5) where the mutant operators might be applied. Mutating statement 3 in *P'* would lead to the generation of three mutants (P_1": sum = a−b, P_2": sum = a∗b, and P_3": sum = a/b), whereas mutating statement 5 in *P'* generates three other mutants (P_4": sum = sum−2, P_5": sum = sum∗2, and P_6": sum = sum+2). Only one mutant (P_5") results in an execution of all test cases in *T* without failure. An important question that needs to be answered is: Do we have to generate each mutant and re-execute all test cases in *T* against it in order to find the fix for the fault?

Supposing we are to follow the above procedure in order of the statement ranking provided by Tarantula, then we would generate the mutants based on statement 5 before those of statement 3 (because statement 5 is at the top of the ranking). Thus, we would be able to find the fix (P_5") earlier and therefore not even need to examine the mutants based on statement 3. In the worst case, we would need to examine mutants P_4" and P_6" before P_5", but in the best case we would do so for P_5" first before any others. Thus, in the worst case we would need to examine three mutants to fix the fault, which is 50% of the total number of mutants. But in the best case, we would only need one mutant to successfully fix the fault, which is only 16.67% of the total mutants that would need to be examined

Thus, the combination of the fault localization process and the mutation technique can potentially not just lead to the automatic fixing of faults, but also do so with reduced overhead. The example that is presented is simplistic and real-life programs can be much larger and more complex. However, such an example has been provided purely for illustrative purposes and to better clarify the fault fixing process.

3. Case Studies

In this section we present an empirical evaluation of our proposed strategy on real programs in order to assess its effectiveness in terms of fixing faults.

3.1. Experimental Design

The fault-fixing strategy proposed in this paper bears with it a certain degree of flexibility and customizability. Certain choices need to be made on how mutation may be applied and different choices may lead to different results.

Mutant Operators Used: The use of a larger and more diverse set of mutant operators may result in the generation of more mutants, but at the same time, may result in the ability to fix more types of faults. Therefore, the choice of which mutant operators are to be used has a strong impact on the effectiveness and efficiency of the strategy. A highly effective strategy shall be able to fix many different kinds of faults, yet a highly efficient strategy will require minimum overhead, i.e., a relatively smaller number of mutants will need to be examined in order to fix the faults. For the purposes of this paper, two classes of mutant operators are applied towards fixing the faults (a subset of those used in [2]):

- Replacement of an arithmetic, relational, logical, increment/decrement, or assignment operator by another operator from the same class.
- Decision negation in an *if* or *while* statement.

Further discussion on the relevance of the choice of mutant operators used is presented in Section 4.1

Identifying a mutant as a potential fix: Since this is a controlled experiment, the correct versions of each of the programs under study are available. Therefore, a mutant can be classified as a fix if it matches the correct version of the program, and any mutant which does not match the correct version can be classified as not-a-fix. This is achievable via string matching. In other words, there is no need to execute the entire test set for each program against its mutants to observe which mutants are killable and which are not. However, in practical settings the correct version of the program is not available. To emulate such a scenario, we decide to classify a mutant as a *potential fix* if the mutant is live, (i.e., un-killable with respect to the new requirement to kill a mutant presented in Section 2.3) by any of the test cases in the available test set.

The stopping criteria for mutant examination: Let us suppose that the total effort it takes to execute a mutant against all test cases in a given test set *T* is represented by *e*. Let us also assume that it takes us *n* mutants to successfully find a potential fix, i.e., we must go through (execute all test cases against) *n*-1 non-fixes before we find the fix. Then, the effort spent to produce the fix is given by

$$\text{Effort } (E) = ne$$

We emphasize that this equation does not represent all the effort spent in detecting a fault, localizing it, generating and executing the corresponding mutants, and then producing a guaranteed fix. Instead it only represents the effort spent to execute all test cases in *T* against the mutants until a fix is found (namely, a mutant against which all test cases in *T* are successful). Thus, one of the important variables that determines this effort is *n*.

But we cannot guarantee that a faulty program is fixable using the proposed strategy. This depends on the relationship between the types of faults we are trying to fix and the set of mutant operators that is to be used. Therefore, when we find ourselves unable to fix a program after the generation of a certain number of mutants, we are left with a choice: continue, reduce, or stop immediately the mutant generation and execution process. The first choice, continue, implies that unless a fix is found the process will not stop until all possible mutants (based on the mutant operators used) for all statements in the program have been examined. This can consume a significant amount of resources. For a large program where the total number of mutants may also be very large, the first choice is not practical. It is reasonable to have some sort of stopping criteria, such that if we do not find the fix within a certain threshold, we should stop trying to fix the program using the proposed strategy. That is, we should place a restriction on the effort spent in fixing a faulty program.

In this paper we present a possible threshold in terms of the percentage of code that has been considered (namely, the percentage of statements based on which mutants have been examined). Data according to such a criterion is provided in Section 3.4. Another possible criterion that might be considered

is to place an absolute cap on the maximum number of mutants that are to be examined in order to find a fix. However, in practice we do not recommend the use of such a criterion because the number of mutants examined is directly related to the choice of mutant operators used. If one employs a large set of mutant operators, then a relatively larger number of mutants might need to be examined per statement, and we may saturate our threshold very quickly. Also, it may be difficult for programmers and managers to estimate a good size for such a cap. Therefore, we suggest that should such a criterion be employed in practice, then it must only be done so after a careful and comprehensive analysis. However, for the readers that might be interested, we present the total number of mutants that require examination, in the case of the studied programs that are fixable using the proposed strategy and our selected mutant operators.

It is very important to note that the number of mutants examined to fix a fault (if fixable) also depends on the number of statements that need to be examined to find the fault. This in turn depends on the quality of the fault localizer. A good fault localizer should place the suspicious statement towards the top of its ranking, and therefore, when fixing a fault (if fixable) the mutant representing the fix can be generated based on the statements at the top of the list. Further discussion on this is presented in Section 4.2.

3.2. Subject Programs

Two sets of subject programs (eight programs in total) were used in our study – the seven C programs of the Siemens suite and a JAVA-based Ant program. Thus, experiments have been performed on both C and JAVA programs, but the mutant operators used by us (see the ones listed in Section 3.1) are the same regardless of the programming language.

Siemens Suite: The seven C programs of the Siemens suite have been well studied and used in several fault localization studies [8,17,19,24,31,36]. The correct and faulty versions of each of the programs, as well as the respective test sets were downloaded from [28]. Even though 132 faulty versions were downloaded, faults that did not result in any test case failure were excluded. This is because the Tarantula fault localizer requires the failure of at least one test case in order to assign meaningful suspiciousness values to the statements. Faulty versions where the faults were located in the header files were also excluded because coverage information could not be properly recorded for the header files. Altogether three faulty versions were excluded from our study: version 9 of the program *schedule2*, and versions 4 and 6 of the program *print_tokens*; leaving us with a total of 129 usable faulty versions.

Ant: Ant is a Java-based build tool supplied by the open source Apache project [13]. The source code for version 1.6beta of Ant as well as a set of 6 single-fault versions and 877 test cases were downloaded from [15]. Of the 877 test cases, six of them could not be executed in our environment (please refer to Section 3.3 for a full description of our execution environment). Hence, these six test cases were discarded from our study. A summary of the number of faulty versions and test cases of each program used in our experiments is presented in Table 1. Information on the size of each program in terms of the number of lines of code (LOC) in the correct version is also provided. The Siemens suite programs allow us to evaluate the proposed strategy on programs of a relatively smaller size, while the Ant program allows us to do the same on a program of a much larger size (in terms of the LOC metric).

Table 1: Summary of each program

Program	Number of faulty versions	Number of test cases	LOC[5]
print_tokens	5	4130	539
print_tokens2	10	4115	489
schedule	9	2650	397
schedule2	9	2710	299
replace	32	5542	507
tcas	41	1608	174
tot_info	23	1052	398
Ant	6	871	75333

We note however, that all of the faulty programs used in this study are categorically single-fault programs (i.e., each faulty version has exactly one fault in it). There are two reasons why we decide to restrict ourselves to only using single-fault programs. First, since this is an initial study, we want to validate whether the proposed strategy is practical and effective on single-fault programs before we move on to multiple-fault programs. Second, were we to use programs with multiple faults, it might introduce ambiguity in our results as some faults in a multiple-fault program may be fixable by our selected mutant operators, while others in the same program may not. As a result, it is difficult to claim whether such a program is fixable by the proposed strategy. This also implies that the choice of which multiple-fault programs should be used in our study becomes highly significant. To avoid such complications, we only use the downloaded set of single-fault versions and defer the study on multiple-fault programs to future experiments. A discussion on how the proposed strategy may be applied to programs with multiple faults is presented in Section 4.4.

3.3. Data Collection

During the fault localization step, each faulty version was executed against all its corresponding available test cases and the success or failure of an execution was determined by comparing the outputs of the faulty version and the correct version of the corresponding program. A failure was recorded whenever there was deviation of an observed output (output from a faulty version) from the expected output (output from the correct version of the program).

To collect coverage information for the C programs of the Siemens suite, we made use of χSuds [37] which has the ability to record which statements are covered by a test case, even in the event of segmentation faults. In order for a statement to have been covered by a test case we require that it must have been executed by the test case at least once. In the case of the Ant program, which is JAVA-based, Clover [14] was used to instrument the code and collect coverage information. Similar to the approach used in [17], only executable statements (see Footnote 3) are considered. All multi-line statements are combined into a single source code line such that they are counted as a single executable statement.

When trying to identify a fix, each mutant of a faulty program was executed against all of its corresponding available

[5] The LOC refers to the size of the program, ignoring blank lines and comments, but including both executable and non-executable statements such as function and variable declarations..

test cases. As per the discussion in Section 2.3, the success or failure of an execution was determined by comparing the outputs of the mutant and the correct version (rather than the faulty version) of the program. A failure implies that the mutant does not represent a potential fix.

All program executions were on a PC with a 2.13GHz Intel Core 2 Duo CPU and 8GB physical memory. The compiler used for the Siemens suite programs was GCC 3.4.3, while for the Ant program the JAVA compiler version was 1.5.0_06. All mutants were generated programmatically via the combination of scripts and JAVA programs.

3.4. Results

The proposed strategy can be evaluated across two aspects: first, how effective mutation is at producing fixes for faults; and second, when a fault is fixable, the number of mutants that need to be examined in order to find a fix.

3.4.1. Effectiveness at Fixing Faults

The effectiveness of our fault-fixing strategy can be expressed in terms of the percentage of faults that are fixable, i.e., the percentage of faulty versions for which mutation is able to produce the correct version of the program. This information is presented on a program-by-program basis in Table 2.

Table 2: Number of faults fixed by using our strategy

Program	Percentage of Faults Fixed	
print_tokens	**0%**	(0/5)
print_tokens2	**10%**	(1/10)
schedule	**0%**	(0/9)
schedule2	**11.11%**	(1/9)
replace	**9.38%**	(3/32)
tcas	**21.95%**	(9/41)
tot_info	**34.78%**	(8/23)
Ant	**50%**	(3/6)
All	**18.52%**	(25/135)

Based on the results in the table we find that collectively (across all of the programs), the proposed strategy can automatically produce the correct fix for 18.52% of the faulty versions. However, we also note that the fraction of faults fixed does not seem to be uniform across all of the programs. More faulty versions can be fixed in some cases, while for certain programs none of the faults can be fixed automatically. Across all of the programs of the Siemens suite we find that a total of 17.05% (22/129) of the faults can be fixed automatically. In contrast 50% (3/6) can be fixed in the case of the Ant program. The tcas program of the Siemens suite has the highest percentage of faults fixable (21.95%).

The reason for such differences in the ability of the proposed strategy to fix faults lies in the nature of the faults themselves and the choice of mutant operators used. For example, in the case of the program print_tokens, 4 of the 5 faults involve re-locating, deleting or inserting new program statements. The remaining fault is due to a replacement of the value '80' with the value '10' in a variable assignment. We are unable to fix these faults automatically because the mutant operators we used (see Section 3.1) do not handle such faults. Further discussion on this, and why certain faults are fixable while others are not, is presented in Section 4.1.

3.4.2. Efficiency in Fixing Faults

The analysis presented in Section 3.4.1 suggests that mutation can be used to fix faults. However, it does not address the question of effort that needs to be spent in order to fix the faults automatically. The discussion on the 'stopping criteria for mutant generation and execution' (see Section 3.1) explores such effort. However, before we present the data, there are some important issues that need to be covered. We recall that the Tarantula fault localizer provides us with two levels of effectiveness – the *best* and the *worst*. Therefore, we also correspondingly have two different values as to the number of mutants that must be examined in order to fix a fault – the best case and the worst case. In the best case, the mutant which represents a fix is examined based on a faulty statement that is examined before all other statements of the same suspiciousness. In the worst case the mutant which represents a fix is examined based on a faulty statement which is examined after all statements of the same suspiciousness.

In fact because each statement may result in the generation of multiple mutants, the best and worst cases can be further divided as *best-of-the-best*, *worst-of-the-best*, *best-of-the-worst*, and *worst-of-the-worst*. In the best-of-the-best case, not just is the faulty statement examined first, but among all mutants generated based on that statement, we examine the mutant that corresponds to the fix first. In the worst-of-the-best case, the faulty statement is examined first, but the mutant that is the fix is examined last. Similarly, we have a best-of-the-worst case and a worst-of-the-worst case. In our study, we take a conservative approach by using the worst-of-the-best as the *best* case and the worst-of-the-worst as the *worst* case. However, since the number of mutants generated per statement using our proposed mutant operators is not expected to be a very large number, the difference between the *best-of-the-best* and the *worst-of-the-best* is small. The same also applies to the difference between the *best-of-the-worst* and the *worst-of-the-worst*. Figure 3 shows the number of mutants that need to be examined for the 25 fixed faults in Table 2.

In Figure 3 the *y*-axis represents the number of mutants examined (non-cumulative) while the *x*-axis represents the number of faults that can be fixed by examining a number of mutants less than the corresponding *y*-value. From the figure we observe that for most of the faulty versions (21/25), the faults can be fixed by examining less than 500 mutants, irrespective of whether the best case or worst case is considered. Even if we restrict ourselves to examining only 50 mutants, we can still fix 10 faults in the best case and 9 in the worst, which is highly efficient. The maximum number of mutants that is required to fix a fault is 2054 in the best case and 2876 in the worst case, and this particular fault is part of the Ant fault set. In fact, all three of the faulty versions that require the examination of more than 1000 mutants are from the Ant program. The reason for such an observation is that the Tarantula fault localizer is not very effective at localizing these faults and therefore, the fault-containing statements are not at the very top of the corresponding rankings. Hence, several non-faulty statements need to be examined, and their corresponding mutants examined, before the fault-fixing mutant is found. Every fault in the Siemens suite that is fixable (using the selected set of mutant operators) can be fixed by examining a maximum of 737 mutants in both the best and worst cases. Placing a cap on the

total number of mutants that are to be examined per faulty version is therefore one possible criterion to stop the fault-fixing process, but its use is not recommended without a careful analysis for the reasons discussed in Section 3.1.

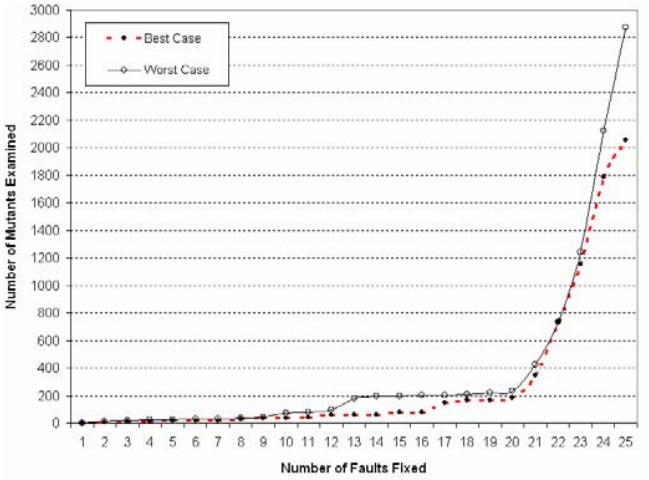

Figure 3: Number of mutants that need to be examined in order to fix the faults

Another possible stopping criterion can be set in terms of the maximum percentage of statements to which we can apply the mutant operators. We partition the percentage of statements into intervals of 5% and present the corresponding number of mutants examined, as well as the total number of faults fixed for the 129 faulty versions of the Siemens suite, in Table 3.

Table 3: Percentage of statements and the number of mutants examined as well as the number of faults fixed for the Siemens suite

Percentage of statements examined	Best Case		Worst Case	
	Number of mutants examined	Number of faults fixed	Number of mutants examined	Number of faults fixed
5%	4132	9	4435	9
10%	8726	11	9236	10
15%	13126	15	14480	11
20%	17554	15	19853	11
25%	22102	16	25301	13
30%	27652	16	30427	15
35%	31012	18	34593	15
40%	35671	21	39654	18
45%	40301	21	44133	19
50%	42764	21	48838	19
55%	48736	21	53547	19
60%	51492	21	58047	19
65%	56839	21	62370	19
70%	61892	21	67310	19
75%	66849	21	71990	21
80%	70012	21	76618	21
85%	74332	22	81198	21
90%	78194	22	85955	22
95%	81987	22	89673	22
100%	82829	22	91158	22

Such data help us understand the expense of using the proposed strategy from a different perspective. For example, examining 5% of the statements (starting from the top of the ranking based on the suspiciousness provided by Tarantula), we can fix 9 of the 129 faults by examining 4132 mutants in the best case and 4435 in the worst. Data for the Ant program is not provided in Table 3 due to the sheer size difference between Ant and the programs in the Siemens suite in terms of the number of executable statements. The correct version of Ant has 75,333 lines of code (see Table 1) and 45,701 executable statements, whereas the largest number of executable statements for the programs in the Siemens suite is just 216 (for the *replace* program) which is less than 0.473% that of Ant. Thus, the stopping criteria for the Ant program need to be evaluated at a different and finer granularity than that of the Siemens suite. As an example, by examining only 1% of the executable statements we are able to fix 3 of 6 faults using a total of 3812 mutants in the best case and 4347 mutants in the worst case. Due to space constraints the details for the Ant program are not provided here. Interested readers may get such information by contacting the authors.

4. Discussion

In this section we discuss some of the issues relevant to the proposed strategy, and the experiment results, that have been presented in the paper.

4.1. Mutant Operators and Fault Fixability

The number of faults that are fixable by the proposed strategy depends to a large degree on the choice of mutant operators. The larger the set of mutant operators used, the larger the number of mutants generated per statement may be; yet the more types of faults that may possibly be fixed. For example, we find that by adding the *off-by-one* mutant operator (different from the increment/decrement operator used in this study) for scalar values to our selected set, we are able to fix 27 faults of the Siemens suite as opposed to the earlier 22. Since this was an initial study and we wished to first verify the effectiveness of our strategy before undertaking complex studies, we restricted ourselves to a simple set of mutant operators.

However, there may be certain faults that are very difficult, if not entirely impossible, to fix using any mutant operators. One such class of faults includes those that involve missing code. When a statement is missing from a program, and the lack of that statement is what is causing the program to be faulty, even if the location of the missing statement can be identified, it is nearly impossible to insert the missing statement by using mutation alone. In fact, without additional information, there may potentially be an infinite number of candidates to substitute the missing statement.

Overall, the fixability of a fault using our proposed strategy depends on its *modus operandi*, i.e., how the fault is manifested in the source code. In practice, the choice of which mutant operators to use would come down to the intuition, needs, and resources of the programmers and software engineers applying the strategy.

4.2. The Importance of the Fault Localizer

This paper proposes the use of mutation to automatically suggest fixes for faulty programs, but also presents a novel combination of mutation and fault localization in order to reduce the overhead involved. Assuming a fault is fixable by the selected set of mutant operators, we can find the mutant corresponding to the fix faster, if the fault localizer pushes the faulty statement (which is mutated to find the fix) towards the top of its ranking. The

more effective the fault localizer, the more efficient the proposed strategy is expected to be. For the purposes of this experiment, we use the Tarantula fault localizer [17]. Several studies [31,35, 36] have presented alternative methods that empirically prove to be more effective (by examining less code) than Tarantula at localizing faults; and the use of such localizers might very well improve our results especially in terms of reducing the number of mutants that require examination to fix faults. However, the purpose of this study was never to compare the effectiveness of fault localizers with one another and instead only aimed to demonstrate the feasibility of using the proposed strategy. The choice of Tarantula was motivated by the fact that it is simple in design and works reasonably well in most cases. However, our strategy is not specifically limited to any particular fault localizer. Others who wish to apply the proposed strategy are welcome to use any fault localizer of their preference.

4.3. The Proposed Strategy and Test Set Quality

Debugging typically takes place after observing one or more failures when the program is executed against test cases in a given test set *T*. When applying the proposed strategy, a potential fix is identified when all test cases in *T* execute successfully against a mutant. However, if there are multiple such mutants then it is possible that

a) these mutants are functionally equivalent to the correct program, or
b) *T* is not good enough to distinguish the behavior of these mutants from the correct program. This includes a possibility such that none of these mutants is the correct version of the program.

The observance of case (b) suggests *T* is an inadequate test set and should be augmented. On the other hand, if we are able to fix the fault and we have only one such fix (i.e., only one mutant leads to all test cases in *T* executing successfully) then it increases our confidence in the goodness of *T*. Thus, the proposed strategy does not just allow us to produce potential fault fixes automatically, but also helps us assess the goodness of the test set being used. In our experiments, for none of the programs under study did we encounter the case where a mutant resulted in all the test cases executing successfully, yet the mutant was not the correct version of the program. This suggests that the downloaded test sets for the studied programs were adequate, at least as far as the mutants that are generated using the selected set of mutant operators are concerned.

4.4. Programs with Multiple Faults

The experiments that have been performed in this study have all been on single-fault programs. Here we discuss how the strategy may be extended to handle programs with multiple faults as well.

When analyzing programs with multiple faults we note that the failures of two test cases are not necessarily linked to the same causative fault. One solution to overcome this problem is to produce *fault-focused* clusters [16] by using appropriate techniques (which are beyond the scope of this paper) to group failed test cases related to the same fault into the same cluster. Then, a selected fault localization technique using the failed test cases in each cluster and some successful test cases can be applied to locate the fault that the cluster is linked to. Assuming the fault localizer is of good quality, the statement rankings produced by the application of the localizer to each cluster should each have their causative faulty statements towards the top of their rankings. At this point, the selected mutant operators may be applied to produce mutants based on each ranking.

For discussion purposes, let us assume we have a program with two faults (f_α and f_β). We can generate two statement rankings $\mathcal{R}_\alpha$ and $\mathcal{R}_\beta$ using failed test cases in the fault-focused clusters for f_α and f_β, respectively, and some successful tests. Then we generate mutants with respect to the top statement of $\mathcal{R}_\alpha$ and $\mathcal{R}_\beta$, and denote them as $\mathcal{M}_{\alpha,1}$ and $\mathcal{M}_{\beta,1}$. We can try to fix f_α and f_β simultaneously by producing a higher-order mutant which is a combination of a mutant from $\mathcal{M}_{\alpha,1}$ and a mutant from $\mathcal{M}_{\beta,1}$. Different combinations are possible by using different mutants from $\mathcal{M}_{\alpha,1}$ and $\mathcal{M}_{\beta,1}$. Should the program still not be fixed after trying all possible combinations, we may generate more mutants with respect to the second statement from the top of $\mathcal{R}_\alpha$ and $\mathcal{R}_\beta$, and denote them as $\mathcal{M}_{\alpha,2}$ and $\mathcal{M}_{\beta,2}$, respectively. But then are mutants in $\mathcal{M}_{\alpha,2}$ to be combined with those in $\mathcal{M}_{\beta,1}$ or those in $\mathcal{M}_{\beta,2}$? How about the combination of mutants in $\mathcal{M}_{\beta,2}$ and $\mathcal{M}_{\alpha,1}$? There are different ways to combine these mutants and to discuss which one gives better results requires substantial further research, which is beyond the scope of this paper.

The above approach can be easily extended for programs with n faults, where $n > 2$. The general idea is that a program with two faults may be fixed by combining two different mutants each of which fixes one fault as described above. Similarly, an n-fault program might be fixed by an n-tuple with each element as a mutant to fix one of the faults. This is consistent with the idea that a faulty program with n faults is essentially n mutations away from the correct program. Therefore, it is reasonable to apply the strategy by using a combination of n different mutants as described above to produce the potential fixes for programs with n faults.

4.5. Programs with a Single fault on Multiple Lines

A single fault may span multiple lines that are not necessarily contiguous and may spread across multiple functions (or classes in the case of object-oriented programs such as JAVA and C++). In order for a programmer to fix such a fault, he/she must consider all of the lines that are related to this fault as addressing just a single line may not fix the fault in its entirety and may still result in program failure. The proposed strategy in its current form applies to programs with individual faults that are only on a single line. This is because we currently restrict ourselves only to first order mutation (see Section 2.2) and therefore, can only fix programs where the correct version is exactly one syntactic change away from the incorrect version. A fault that spans multiple lines may need to be fixed by a combination of several mutants similar to the strategy presented in Section 4.4.

4.6. Threats to Validity

There are several threats to the validity of the proposed strategy, and its accompanying results, which include but are not limited to the following.

The primary threat is that the proposed strategy is based on the premise that mutation-based faults represent realistic faults [10,19,21,22] and therefore, realistic faults may also be fixed by mutation. However, we never claim that every fault may be fixed by the application of some mutant operators. It is possible that the fault still remains unfixed after using the proposed strategy.

However, arguably even if we cannot tell programmers how to fix a fault; we can still give them information on how *not to fix* a fault. A programmer trying to fix a fault can avoid attempts that have already been tried via the proposed strategy. Thus, what is provided to the programmer is more of a report than just a ranked list of suspiciousness statements generated by using a fault localizer, as it also contains information on mutants that have been tried and have failed as potential fixes. This additional information can provide a programmer with some intuition on how to actually fix the fault and save his/her time. Another point to be careful about is throughout the paper we emphasize that the proposed strategy (in its current form) should be used to automatically produce *potential* fixes. It is possible for a mutant to result in a successful execution of all test cases in a given test set *T*, but still not be a correct fix. Therefore, the determination of a suggested fix as correct or incorrect and the actual use of such a fix is still left up to the programmer's discretion.

A third threat that may limit the generalization of our results is the choice of subject programs. Since this is an initial study, experiments have only been performed on two sets of programs (8 programs corresponding to 135 faulty versions in total). However, efforts have been made to diversify in that the Siemens suite was selected because it has been widely used in many studies, while the Ant program was selected to contrast the small size of the Siemens suite programs. Additionally, while our experiments have been performed on both C programs (Siemens suite) and a JAVA program (Ant), the same set of mutant operators (see Section 3.1) was used for both. Furthermore, all of the programs, faulty versions, and test sets were downloaded from a public repository [15,28]. Thus, none of the faults have been seeded by us and there is no bias towards a particular type of fault that may be relatively easier to fix.

Finally, while checking whether a mutant is a potential fix, the proposed strategy requires the mutant to be executed against all test cases in *T*. This process continues until we either find a potential fix, or we run out of mutants. An important assumption made here is that the cost for executing all test cases in *T* on a mutant is relatively low. Should this not be the case then a possible solution would be to execute only a subset of test cases in *T* against the mutant or perhaps order the test cases in a certain way such that test cases are capable of killing the most mutants are executed first. How to decide which subset of test cases should be selected or how to identify an optimum ordering of test cases is out of the scope of this paper. Interested readers may refer to papers such as [11,25,26,32,33].

5. Related Studies

To the best of our knowledge, we are not aware of any other studies that propose the use of mutation to automatically suggest fixes for software faults, and demonstrate the strategy in the same manner as has been presented in this paper.

The authors of [27] present an approach for combined fault localization and correction for sequential systems provided the specification is given in linear-time temporal logic. Their approach is based on infinite games and a fix occurs when a component is replaced by an arbitrary new function which is valid for all possible input sequences. However, their approach is restrictive in that it is limited to linear-time temporal logic-based specifications and the fixes are limited to only Boolean functions which is not the case for the strategy proposed in this paper.

In [7] the intrinsic redundancy of some complex software systems to provide the same results in multiple ways is exploited as a way to automatically recover from functional failures. An execution sequence that is equivalent to a sequence that results in failure, is searched for and if found, is used as an alternative to the failing sequence. However, the approach presented in [7] only applies to models that are derived from software specifications and not the source code. Furthermore, the approach only helps identify an alternate way to produce a correct result without addressing the fault(s). The literature on software and hardware reliability is also abundant with fault tolerance and fault recovery techniques such as the ones in [6]. However, they only ensure that a system can handle or recover from a failure, without attempting to fix the cause of the failure.

More in the direction of self-healing code, the authors of [12] propose a combination of aspect-oriented programming, program analysis, artificial intelligence, and machine learning techniques for the fault diagnosis and self-healing of interpreted object-oriented applications. However, the applicability of their approach is only limited to large self-aware systems and as the authors point out, self-healing at the code level is very complex as the system must derive intended behavior of the system. In [3] the authors propose an evolutionary approach to automate the fixing of bugs that is based on Genetic Programming (GP), co-evolution and search based software testing. An extended form of genetic programming is also used in [30] to evolve program variants when a fault is discovered, such that the variant retains functionality, but avoids the defect in question.

6. Conclusions & Future Work

This paper takes a step in the direction of automated debugging by proposing a strategy to automatically produce potential fixes for software faults. This is done by combining the mutation technique and fault localization. The proposed strategy is evaluated against the seven programs of the Siemens suite and the Ant program. Results indicate that the mutation technique is effective at producing potential fixes to faults automatically (18.52% of the faults are fixed without any human intervention). Furthermore, combining this approach with fault localization allows us to significantly reduce the overhead involved. Even when the strategy is not able to fix some faults, a report is provided to programmers that includes a list of attempted fixes. Therefore, more information is given than just a ranked list of suspicious statements by using a fault localizer alone.

Future work includes, but is not limited to, evaluating the strategy against a broader set of subject programs and observing the effectiveness with different and more diverse sets of mutant operators. We also wish to extend the effectiveness of the fault-fixing strategy to programs with multiple faults and faults that span multiple lines. Exploring test case prioritization techniques to reduce the overhead involved in mutant examination is also one of the future areas of study.

References

1 H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," *in Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, pp. 143-151, Toulouse, France, October 1995

2 J. H. Andrews, L.C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?", in *Proceedings of*

the 27th International Conference on Software Engineering, pp. 402-411, St. Louis, Missouri, USA, May, 2005

3 A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in Proceedings of the IEEE Congress on Evolutionary Computing, pp. 162-168, Hong Kong, China, June 2008.

4 A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, 1(1):11-33, January 2004

5 T. A. Budd, "Mutation analysis of program test data", *Ph.D. Dissertation*, Yale University, 1980

6 M. Butler, C. B. Jones, A. Romanovsky and E. Troubitsyna (Editors), "*Methods, models and tools for fault tolerance*", Springer-Verlag, 2009.

7 A. Carzania, A. Gorla and M. Pezze, "Healing web applications through automatic workarounds," *International Journal on Software Tools for Technology Transfer*, 10(6):1433-2787, December 2008.

8 H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th Internationa*l *Conference on Software Engineering*, pp. 342-351, St. Louis, Missouri, USA, May 2005

9 R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," in *IEEE Computer*, pp. 34-41, 1978

10 H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, 32(9): 733-752, September 2006

11 S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies" in *IEEE Transactions on Software Engineering*, 28(2): 159-182, February 2002.

12 A. R. Haydarlou, B. J. Overeinder and F. M. T. Brazier, "A self-healing approach for object-oriented applications," in *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*, pp. 191-195, Copenhagen, Denmark, August 2005.

13 http://ant.apache.org

14 http://www.atlassian.com/software/clover/

15 http://sir.unl.edu/portal/index.html

16 J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 16-26, London, UK, July 2007

17 J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th International Conference on Automated Software Engineering*, pp. 273-283, California, USA, November 2005

18 B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15-26, Chicago, June 2005

19 C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, 32(10):831-848, Oct. 2006

20 G. J. Myers "*The Art of Software Testing*" *2nd Edition* (Revised and Updated by T. Badgett and T. M. Thomas, with Cory Sandler), John Wiley & Sons, 2004

21 A. S. Namin, J. H. Andrews, and Y. Labiche, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, 32(8):608-624, August 2006

22 A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, 1(1): 3-18, January 1992

23 A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, 5(2): 99-118, April 1996

24 M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30-39, Montreal, Canada, October 2003

25 G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites", in *Proceedings of the International Conference on Software Maintenance*, pp. 34-43, Bethesda, Maryland, USA, November 1998.

26 G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Test case prioritization" in *IEEE Transactions on Software Engineering*, 27(10): 929-948, October 2001

27 S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," in *Proceedings of the Conference on Correct Hardware Design and Verification Methods*, pp. 35-49, Saarbrücken, Germany, October 2005

28 The Siemens suite, http://www-static.cc.gatech.edu/ aristotle/Tools/subjects/, January 2007

29 I. Vessey, "Expertise in debugging computer programs," *International Journal of Man-Machine Studies: A Process Analysis*, 23(5):459-494, 1985

30 W. Weimer, T. Nguyen, C. Goues and S. Forrest, "Automatically finding patches using genetic programming," in Proceedings of the 2009 International Conference on Software Engineering, pp. 364-374, Vancouver, Canada, May 2009.

31 W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *Journal of Systems and Software*, 83(2):188-208, February 2010

32 W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Software-Practice and Experience*, 28(4):347-369, April 1998

33 W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: a case study in a Space application," *Journal of Systems and Software*, 48(2):79-89, October 1999

34 W. E. Wong and A. P. Mathur, "Fault detection effectiveness of mutation and data flow testing," *Software Quality Journal*, 4(1): 69-83, March 1995

35 W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, 19(4): 573-597, June 2009

36 W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A Crosstab-based statistical method for effective fault localization," in *Proceedings of the 1st International Conference on Software Testing, Verification and Validation* (ICST), pp. 42-51, Lillehammer, Norway, April 2008

37 χSuds User's Manual, Telcordia Technologies, 1998

38 A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the International Conference on Machine Learning*, pp. 1105-1112, Pittsburgh, Pennsylvania, June 2006