Proteum/FL: a Tool for Localizing Faults using Mutation Analysis

Mike Papadakis*, Marcio E. Delamaro†, Yves Le Traon*
*Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg
Email: {michail.papadakis, yves.letraon}@uni.lu
†Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, Brazil
Email: delamaro@icmc.usp.br

***Abstract*—Fault diagnosis is the process of analyzing programs with the aim of identifying the code fragments that are faulty. It has been identified as one of the most expensive and time consuming tasks of software development. Even worst, this activity is usually accomplished based on manual analysis. To this end, automatic or semi-automatic fault diagnosis approaches are useful in assisting software developers. Hence, they can play an essential role in decreasing the overall development cost. This paper presents Proteum/FL, a mutation analysis tool for diagnosing previously detected faults. Given an ANSI-C program and a set of test cases, Proteum/FL returns a list of program statements ranked according to their likelihood of being faulty. The tool differs from the rest of the mutation analysis and fault diagnosis tools by employing mutation analysis as a means of diagnosing program faults. It therefore demonstrates the effective use of mutation in supporting both testing and debugging activities.**

***Keywords*-Mutation analysis; fault localization; software debugging; software testing;**

I. Introduction

Detecting and fixing program faults forms an essential and expensive task of software development. Developers rely on software testing for identifying program flaws and software debugging for diagnosing and removing them. The process of identifying flawed program parts given some failures is usually referred to as fault localization. This process takes place just after identifying program failures and is one of the most expensive activities of software debugging [1].

Mutation analysis forms a powerful technique usually used for driving the testing process. Generally, it seeds some defects into the program under investigation in order to examine its behavior. Considering the testing process, finding artificial defects allows the effective detection of real faults. This practice has been empirically shown to be more effective than using classical test selection criteria (e.g. based on code coverage). In view of this, the present paper explores the idea of using artificial defects as a means of diagnosing real defects. Therefore, mutation analysis is employed in order to guide the fault localization process.

Existing fault localization techniques assist programmers by ranking program locations according to their probability of being responsible for the experienced failures. The underlying idea of these approaches is to compare the similarity between the execution of these code places and the observed failures. Researchers have provided evidence that these approaches are helpful in reducing the cost involved in the debugging activity [2], [3].

Testing using mutation analysis involves designing and executing tests with a set of artificial faults called mutants. The main assumption of the method is that the utilized defects (mutants), despite being artificial defects, behave like real faults. Evidence in support of such a proposition have been given by Andrews et al. [4]. Furthermore, software testing research has successfully demonstrated that detecting mutants results in detecting real faults. Therefore, it can be asked: what is the relation between the mutants' location and the existing faults? Is there any link between the mutants' location with the location of the faults?

The paper presents Proteum/FL a tool that localizes faults using mutation analysis and thus, answering the above questions. The tool was initially used in the work of Papadakis and Le Traon [5] for defining a mutation-based fault diagnosis approach and it can be used to support both testing and diagnosis activities. Previous research on this topic [5] has shown that the implemented approach significantly outperforms the coverage-based fault localization techniques and opens a new direction, mutation-based debugging, to the mutation analysis research.

The remainder of this paper is organized as follows: Section II presents the concepts and details regarding the mutation analysis. Section III and IV respectively introduce fault localization using mutation analysis and describe the implementation details of Proteum/FL. Finally, the relevance of Proteum/FL with other tools and the conclusions made during this research are discussed in Sections V and VI respectively.

II. Mutation Analysis

Mutation analysis injects artificial defects into the program under investigation with the aim of examining its behavior when executed with some test cases. These defects are called *mutants* and they are generated by using a set of simple syntactic rules, called *mutant operators*. Mutants are traditionally utilized to facilitate the testing process here referred to as mutation testing. Proteum [6] is a mutation testing tool that aims at automating the testing process of

978-1-4673-5739-5/13/$31.00

C programs. This section gives a brief description of the mutation testing process, the supported mutant operators and the Proteum mutation testing system.

A. Mutation Testing

Software testing involves the examination of a program by executing a set of test cases. However, in practice there is a need to evaluate the appropriateness of the utilized test sets. Further, in case that the utilized test sets are not adequate, there is a need to guide the design of new test cases. To this end, mutation testing aims at guiding the testers to design sets of test cases and evaluate their adequacy.

Mutation testing requires tests capable of making the seeded defects observable. To this end, a comparison of the programs' outputs is needed in order to decide whether a difference in behavior between the original and the mutant program versions has been triggered. In such cases, the mutants are said *killed*, otherwise they are said *live*. Testing adequacy is measured by the percentage of mutants that are killed by the utilized tests. Unfortunately, not all the mutants can be killed. A mutant for which there is no test data that distinguish its behavior from the original program is said *equivalent*. Therefore, testing adequacy, called *mutation score (MS)*, is measured by the following value:

$$MS = \frac{No.\ killed\ mutants}{No.\ mutants - No.\ equivalent\ mutants}$$

Generally, mutation testing relies on the quality of the involved mutants [7]. The application of the method relies on two hypothesis, the "competent programmer" and the "coupling effect" [7]. The "competent programmer" hypothesis states that the programmers produce programs that are close of being "correct". Thus, only small changes are necessary to effectively exercise the program under test. The "coupling effect" hypothesis states that "Test data that distinguishes all program differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors". Since mutants represent simple faults, the above hypothesis suggests that killing mutants results in revealing both simple and complex faults.

B. Mutation Operators

Proteum employs mutation operators targeting at unit and integration testing faults [8]. The unit-level operators were designed based on the study of Agrawal et al. [9] while the integration testing operators were designed according to the study of Delamaro et al. [8]. Here it should be noted that the focus of the present paper is on locating unit level faults and thus, mutants related to integration testing are not discussed. The employed operators are divided into four main categories (classes): a) STATEMENT b) VARIABLES, c) CONSTANTS and d) OPERATORS. The STATEMENT class contains operators that alter an entire statement or its key syntactic elements. The VARIABLES and CONSTANTS classes contain operators regarding program identifiers and constants, respectively. They model incorrect variable and constant uses. The OPERATORS class contains operators that alter the programming language operator use.

C. Proteum & Proteum/IM

Proteum [6] is among the firstly and most popular mutation testing tools. Originally, it was designed to perform mutation at the unit level by utilizing the mutant operators described in the previous section. It was then extended and named as Proteum/IM, to support both unit and integration testing by utilizing the Interface Mutation [8] approach. A detailed description of the functionality and the implementation of the Proteum/IM tool can be found at [6]. The latest version of Proteum/IM was recently released as an open source software under the "GNU GENERAL PUBLIC LICENSE". It is this version of the tool that it is extended by Proteum/FL and can be found at:

$$http://ccsl.icmc.usp.br/projects/proteum$$

III. Proteum/FL Fault Localization Process

Fault localization is the process of identifying the program places which are responsible for provoking program failures. Typically, fault localization is performed after the testing process. Generally, the testing process involves the design, execution of some test cases and the determination of whether the program behaves as expected. A failure is experienced when the output of a test differs from the expected one, as specified by the tester. The test that results in a failure is called a *failed test*. In the opposite case, the test is called as a *passed test*. To this end, fault localization tries to identify the program places that are responsible for the program failures given a set of failed tests and a set of pass tests. These tests are those used during the testing process. They might have been produced using a test strategy like [10], in a semi-automated way i.e. [11], [12] and [13] or in a manual way. Therefore, given a test suite, fault localization tries to highlight the program statements that are likely to be faulty. Then, the tester will inspect these statements in order to identify the faulty program location.

The underlying idea of most fault localization approaches is to define a suspiciousness metric that measures the probability of a statement to be faulty. The definition of the suspiciousness metric is based on the observation that program failures are the manifestation of faults. Thus, it is natural to expect that failed tests execute program places that correlate with faults. On the contrary, passed tests should execute program places that do not correlate with faults. In view of this, most of the fault localization approaches try to measure the similarity/dissimilarity between the failed/passed tests with the execution of program statements. Among the various similarity measures Proteum/FL uses the Ochiai formula (presented in Table I) [14]. Ochiai is one of the most popular measures and it has been empirically found to

Table I
THE OCHIAI FORMULA

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

Where: **Suspiciousness(e)** is the probability of code element e to be faulty. **totfailed** is the total number of failed tests, **failed(e)** is the number of failed tests that cover (kill) the e code element (mutant) and **passed(e)** the number of pass tests that cover (kill) the e code element (mutant).

be one more effective than some other alternatives. Here, it should be noted that other measures could be used as well. The interested reader can refer to [15] for further details.

Despite the research made in the area of fault localization, existing approaches are far from satisfactory in many situations. This is due to the so-called *coincidental correctness problem* [16]. Coincidental correctness occurs when tests execute the faulty elements but fail to manifest it to a failure. Therefore, the similarity measure between the failed/passed tests with the execution of program statements turns to be missleading. To improve the effectiveness of fault localization, it has been proposed to use mutation analysis [5]. Contrary to structural testing criteria, mutation works with the programs' outputs (it requires the mutants to have an effect to the program output). As a consequence, it can simulate well situations having coincidental correctness.

Fault localization using mutation analysis measures the similarity between the test failed/passed with their respective results on kill/live mutants [5]. To accomplish this, any similarity measure can be used [15]. As pointed out before, Proteum/FL uses the Ochiai formula for this purpose by considering the mutants as the formula elements (e). The aim of the process is to measure the similarity between failures and killed mutants. Thus, in Table I the "execute the e code element" requirement represents a mutant that is killed [5]. Therefore, for each mutant (e) a suspiciousness value is assigned. These values are then ranked in order to get a priority list with respect to decreased suspiciousness values. Since every mutant is created based on syntactic changes, a direct mapping from mutants to program statements can be made. If a mutant alters more than one statements, all these are assigned with the same suspicious value.

The overall fault localization approach is outlined in Figure 1. This approach is implemented in the Proteum/FL tool. The description given in the following section concerns the adaptation of Proteum mutation testing tool (see Section II.B) to effectively localize faults.

IV. PROTEUM/FL

Proteum/FL supports the entire fault localization process using mutation analysis as described in [5]. It automatically performs the generation and execution of mutants and it also reports the suspicious program statement list. This section describes the implementation of the tool, its functionality and some important optimization techniques that it uses.

A. Description

Automatically performing mutation analysis with Proteum/FL requires some parameters to be specified. The user can specify the mutant operators that are going to be used and the way (command or script) that the mutants should be compiled. Additionally, the user need to specify a comparison method that will be used to determine the killed and live mutants. By default, a comparison is made based on the programs resulting values and its printable output. This can be extended by defining a program specific method like comparison of output files etc. The definition can be a perl or shell script. The tool will then automatically do all the required tasks to report the suspicious program statements to the tester.

Proteum/FL has been implemented in Perl and works as a command line tool. It implements the mutants generation, execution and fault localization tasks as described in the next section. The high level architecture is depicted in Figure 2. It consists of three main components: Proteum, the Mutant Generation and the Test Execution Engine components. The Proteum component is actually the command line interface of the Proteum/IM tool. It takes as input a C program and produces the mutants' description file [6]. This file contains information containing the definition of each mutant, its program location, its type and its identification number. This file is then used by the Mutant Generator component in order to generate and compile the sought mutants. To accomplish this task, GCC[1] and Gcov[2] tools are used. Gcov is employed for collecting trace information of the original program and the executable program statements. GCC is used for compiling the mutants. The mutants and the original programs are passed to the Test Execution Engine which performs the execution of the mutants with the available tests and produces a suspicious statement report.

B. Functionality of Proteum/FL

The Proteum/FL tool provides the infrastructure for locating program faults by performing the following steps:

Mutant selection: Proteum/FL applies by default all supported mutant operators. However, it is possible to use a smaller set of mutants or operators. This practice often results in similar results with the default one, with a lower computational cost. Thus, the user can specify a selective set of operators to apply. Alternatively the user can select applying a random percentage of mutants.

Generate mutants: Proteum/FL relies on the Proteum/IM [6] to identify the possible mutants by generating the mutants' description file [6]. Proteum/FL then reads both the source code of the program under test and the mutants'

[1]GNU Compiler Collection, http://gcc.gnu.org/

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

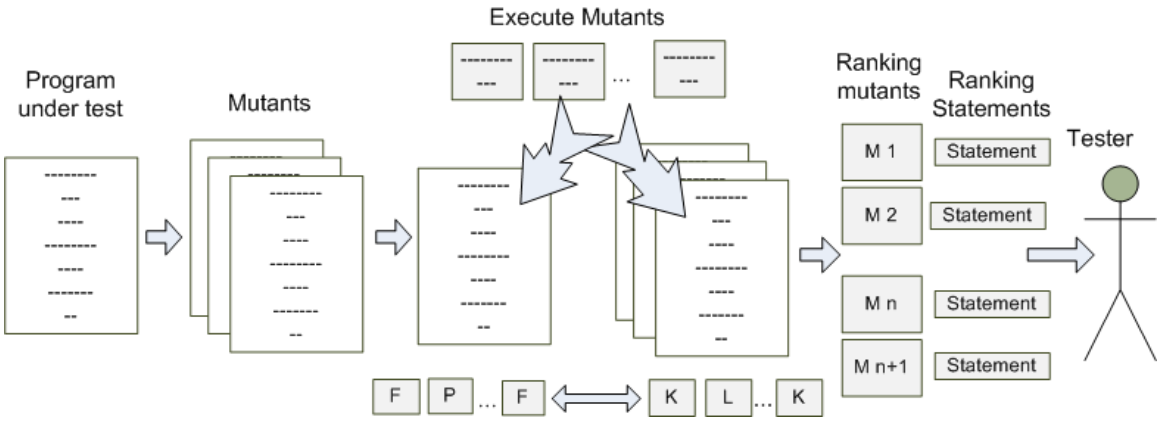

Figure 1. Proteum/FL fault Localization process: Initially, a set of mutants is produced. These mutants are then executed with all the available test cases in order to determine which ones are killed by each one of the utilized test cases. Then a similarity comparison between the failed/passed test cases and the killed/live status of each mutant is performed using the Ochiai formula (I). Then, all the mutants are ranked according to the similarity values computed in previous step. Finally, a ranking list of statements is obtained based on the ranking of mutants and it is reported to the tester.

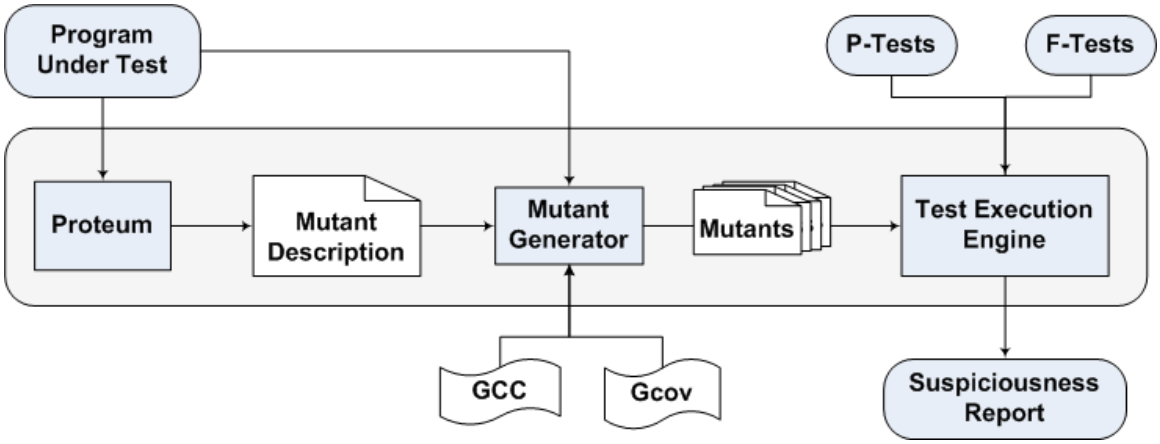

Figure 2. Proteum/FL Architecture

description file and produces the mutants' source code. It actually produces and compiles one source file per mutant. In literature this technique is called *separate compilation* [17] or *compiler-based* [18] technique. Although, more advanced techniques exists like *Mutant Schemata* [7], [17] it was chosen due to its implementation simplicity. Further, by doing so it is possible to construct a test harness. In the *test harness technique* [17] "each mutant is compiled into a shared library that can be dynamically invoked by a test harness". Thus, during the test execution mutants can be efficiently invoked [17] resulting in a reduced execution cost.

Execute mutants: Proteum/FL takes as inputs the failed and passed test cases and executes all the mutants with all tests as required by the mutation analysis technique. It then produces a matrix containing the information of which mutants are killed by each test. This is a computationally expensive part of the process. To efficiently perform it Proteum/FL implements a wide range of optimizations. These are detailed in the next section (Optimizing the Execution of Mutants).

Collect & analyze data: Proteum/FL reads the produced matrices and generates the Suspiciousness Report. This is performed based on the use of the Ochiai formula (Table I).

Rank program statements: Proteum/FL aim is to report the tester a ranked list of program statements. This list assist the tester in locating the program faults by inspecting the most suspicious program statements first. Figure 3 depicts the suspicious statement report produced by the Proteum/FL on a sample program. This report, contains the information about the line of the statements, their suspicious values and the mutants that have these suspiciousness values. It is noted that the statements are ranked according to a decreased suspicious order. Thus, in the report of Figure 3, the most suspicious statement has a suspiciousness value 1.0. This value is assigned to the line 298 based on the suspiciousness of the mutants 1075, 1053 etc.

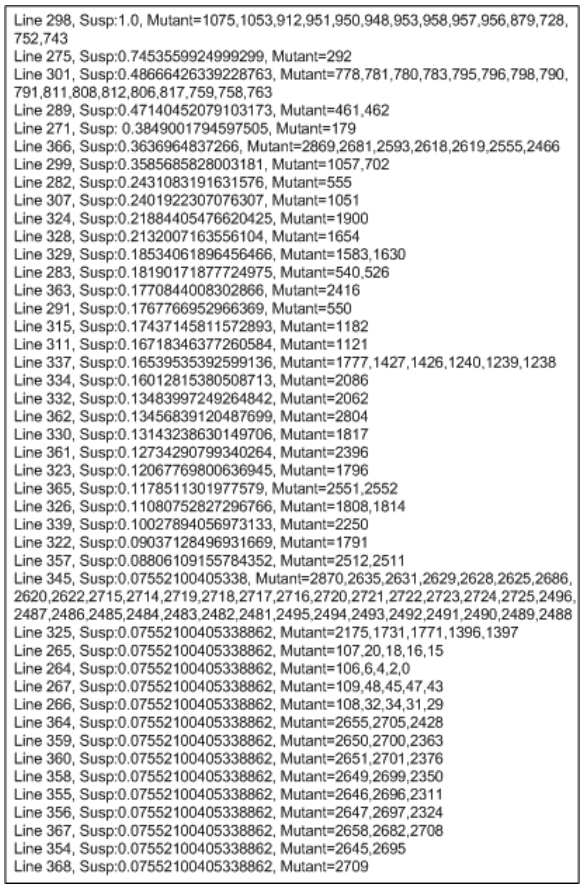

Figure 3. Statement Suspiciousness report

C. Optimizing the Execution of Mutants

Performing mutation analysis requires executing the tests with a huge number of mutants. This process aims at determining the mutants that are killed by each one of the utilized tests and it involves vast computational resources. In order to be efficient, Proteum/FL implements the following optimizations:

Coverage data: Generally, a mutant not executed by a test case has no chance of being killed and thus, its execution results in a waste of time. In view of this, test execution should only focus on mutants that are executed, i.e. test execution reaches the mutated program statement, by a test case. To do it so, Proteum/FL collects execution traces from the original program and executes only those mutants that are reached by each test case.

Parallel Execution: Proteum/FL takes advantage of the parallel and distributed capabilities of the modern computers and thus, it performs a parallel execution of mutants. Since each test is independent of the remaining ones, Proteum/FL assigns one test execution task per utilized process.

Relevant Mutants: Generally, mutants not killed by failed tests do not contribute to the fault localization process and hence they can be ignored. Proteum/FM mutant execution is divided into two stages: the failed tests execution and passed tests execution. It first executes the failed tests and then executes the passed tests by considering the mutants killed by the failed test cases.

The first and second optimizations are usual optimizations implemented in several mutation testing tools. However, none of the existing mutation testing tools targets fault localization which is the main aim of Proteum/FL. The last optimization is dedicated to the fault localization process and thus, there is no other tool doing something similar.

V. Related Work

There is a relatively large number of approaches and tools regarding fault localization. However, there is no other tool to the authors' knowledge that uses mutation analysis to support fault localization.

Mutation analysis is an active research field since 1970s with an increased popularity over the last years [7]. Despite this, only a relatively small number of mutation testing tools exists. Here, a brief description of the most representative tools is given by highlyting their unique characteristics.

As described before, Proteum/IM is one of the first and most popular tools for C programs. It has the main advantage of implementing all the proposed unit [9] and integration [8] level mutation operators. Recently, another tool named Milu [18] has been proposed for C programs. Milu implements a different approach to perform mutation testing based on the notion of higher order mutants. First order mutants are those produced based on one simple syntactic change. Higher order mutants are the combination of one or more first order mutants [7]. Mutation testing tools also exist for the Java programming language. MuJava [17] is one of the most popular tools. Currently it is the only tool that implements specially designed mutation operators for the Java Object Oriented features.

Regarding fault localization, one of the first attempts is Tarantula [2]. Tarantula uses statement coverage informati on to prioritize program statements according to their suspiciousness. Tarantula approach was later extended in order to include various similarity formulas like the Ochiai [14] and [15]. Later, this approach was generalized in order to include other program constructs like program branches [19], definition-use pairs [19] and combinations of them [19]. Other similar tools like Zoltar use Bayesian reasoning [20] for prioritizing suspicious program statements.

VI. Conclusion

The present paper introduced a fault localization tool named Proteum/FL. The innovative part of this tool is the use of mutation analysis for fault localization. The tool supports a comprehensive set of mutant operators specially designed for the C programming language. Additionally, it implements a wide set of optimizations techniques for reducing the mutants' execution cost.

The current version of the tool aims at solving a significant and challenging problem of software debugging. It enables a new direction of research, the use of mutation analysis for supporting software debugging activities like fault localization. We believe that Proteum/FL also opens the way for unifying the testing and debugging activities in a complementary way. In view of this, mutation analysis can be utilized for driving both the testing and debugging processes. This practice merges the application cost of testing and fault localization and results on reducing significantly the debugging effort [5].

Future releases of the tool will integrate recently developed test generation techniques like dynamic symbolic execution [11], [12] search based testing [12] and path based testing [13] towards assisting fault localization [21]. Additionally, the incorporation of the tool with an integrated development environment is also planed.

To learn more about Proteum/FL, refer to:

https://sites.google.com/site/mikepapadakis/proteum-fl

REFERENCES

[1] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459 – 494, 1985.

[2] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*, 2005, pp. 273–282.

[3] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, 2009, pp. 76–87.

[4] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[5] M. Papadakis and Y. Le Traon, "Using mutants to locate "unknown" faults," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*, 2012, pp. 691–700.

[6] M. E. Delamaro and J. C. Maldonado, "Mutation testing for the new century." Kluwer Academic Publishers, 2001, ch. Proteum/IM 2.0: An Integrated Mutation Testing Environment, pp. 91–101.

[7] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[8] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 3, pp. 228–247, Mar. 2001.

[9] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the c programming language," Purdue University, West Lafayette, Indiana, techreport SERC-TR-41-P, March 1989.

[10] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, ser. APSEC '10, 2010, pp. 300–309.

[11] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*, 2010, pp. 121–130.

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

[13] ——, "Mutation based test case generation via a path selection strategy," *Inf. Softw. Technol.*, vol. 54, no. 9, pp. 915–932, Sep. 2012.

[14] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*, 2007, pp. 89–98.

[15] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[16] W. Masri and R. A. Assi, "Cleansing test suites from coincidental correctness to enhance fault-localization," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*, 2010, pp. 165–174.

[17] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[18] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC-PART '08)*, 2008, pp. 94–98.

[19] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, 2009, pp. 56–66.

[20] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: a spectrum-based fault localization tool," in *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime (SINTER '09)*, 2009, pp. 23–30.

[21] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering (ICSE '06)*, 2006, pp. 82–91.