Mathematical Biosciences and Engineering

MBE, 19(6): 6124–6140.
DOI: 10.3934/mbe.2022286
Received: 28 February 2022
Revised: 01 April 2022
Accepted: 06 April 2022
Published: 14 April 2022

http://www.aimspress.com/journal/MBE

Research article

Identification of propagated defects to reduce software testing cost via mutation testing

Dong-Gun Lee and Yeong-Seok Seo*

Department of Computer Engineering, Yeungnam University, Gyeongsan 38541, Republic of Korea

* **Correspondence:** Email: ysseo@yu.ac.kr; Tel: +82538103534; Fax: +82538104630.

Abstract: In software engineering, testing has long been a research area of software maintenance. Testing is extremely expensive, and there is no guarantee that all defects will be found within a single round of testing. Therefore, fixing defects that are not discovered by a single round of testing is important for reducing the test costs. During the software maintenance process, testing is conducted within the scope of a set of test cases called a test suite. Mutation testing is a method that uses mutants to evaluate whether the test cases of the test suite are appropriate. In this paper, an approach is proposed that uses the mutants of a mutation test to identify defects that are not discovered through a single round of testing. The proposed method simultaneously applies two or more mutants to a single program to define and record the relationships between different lines of code. In turn, these relationships are examined using the defects that were discovered by a single round of testing, and possible defects are recommended from among the recorded candidates. To evaluate the proposed method, a comparative study was conducted using the fault localization method, which is commonly employed in defect prediction, as well as the Defects4J defect prediction dataset, which is widely used in software defect prediction. The results of the evaluation showed that the proposed method achieves a better performance than seven other fault localization methods (Tarantula, Ochiai, Opt2, Barinel, Dstar2, Muse, and Jaccard).

Keywords: software testing; mutation testing; software cost; test suite; fault localization

1. Introduction

In software engineering, the maintenance stage accounts for the greatest cost out of all software

development stages [1]. Corrective maintenance is a discipline that deals with finding and fixing the defects contained in software [2–4]. The testing stage is used to determine whether the software contains any defects, and is a research area that has been studied for the longest time in the field of maintenance [4–8]. Testing consists of unit, integration, and system testing [8–12]. Unit testing, which is first conducted to detect defects in specific modules, is generally applied through the following steps [12–14].

1) Choosing the target function to be tested.
2) Creation of test cases for the target function and building a test suite.
3) Conduction of tests based on the test suite.
4) Dealing with the target function according to the test results (test case pass/fail).

The testing is finished or returns to step 1) based on the test results. Figure 1 shows a visualization of the unit testing process.

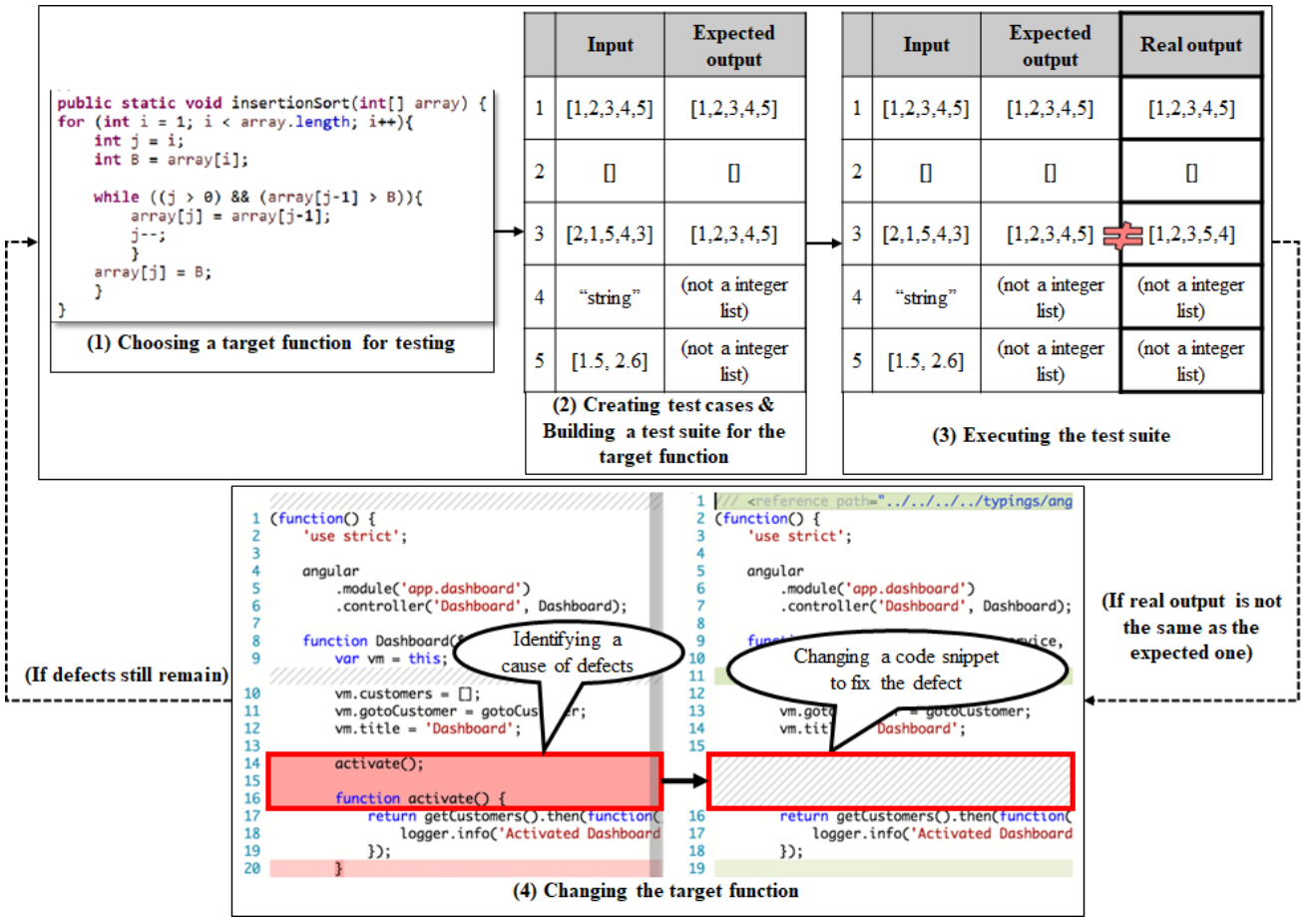

Figure 1. Overview of unit testing.

Even when testing a single function, multiple test cases must be conducted, and numerous functions need to be applied in a single project [15,16]. For this reason, the size of the test suite becomes extremely large, and testing is generally quite expensive because it must be conducted based on the test suite [17,18]. It is therefore necessary to use test cases that properly identify dangerous defects starting when the initial testing is conducted [19–23]. However, even if the performance of the test cases is improved, it still cannot be guaranteed that all defects will be discovered by an initial unit

testing (single round of testing) [24–29]. The following assumptions are therefore made.

- *Original code and test case (Code A)*

The original code is a perfect code without defects, and test cases exist for verification.

- *Code including a single defect (Code B)*

In this case, a single defect is included in code A. The defect then becomes a problem, and the test case that uses code A cannot be passed.

- *Code including two defects (Code C)*

This is the code containing another defect in code B. In other words, two different defects are present. However, unlike the test results of code B, code C passes the test case (the test results are the same as those of code A).

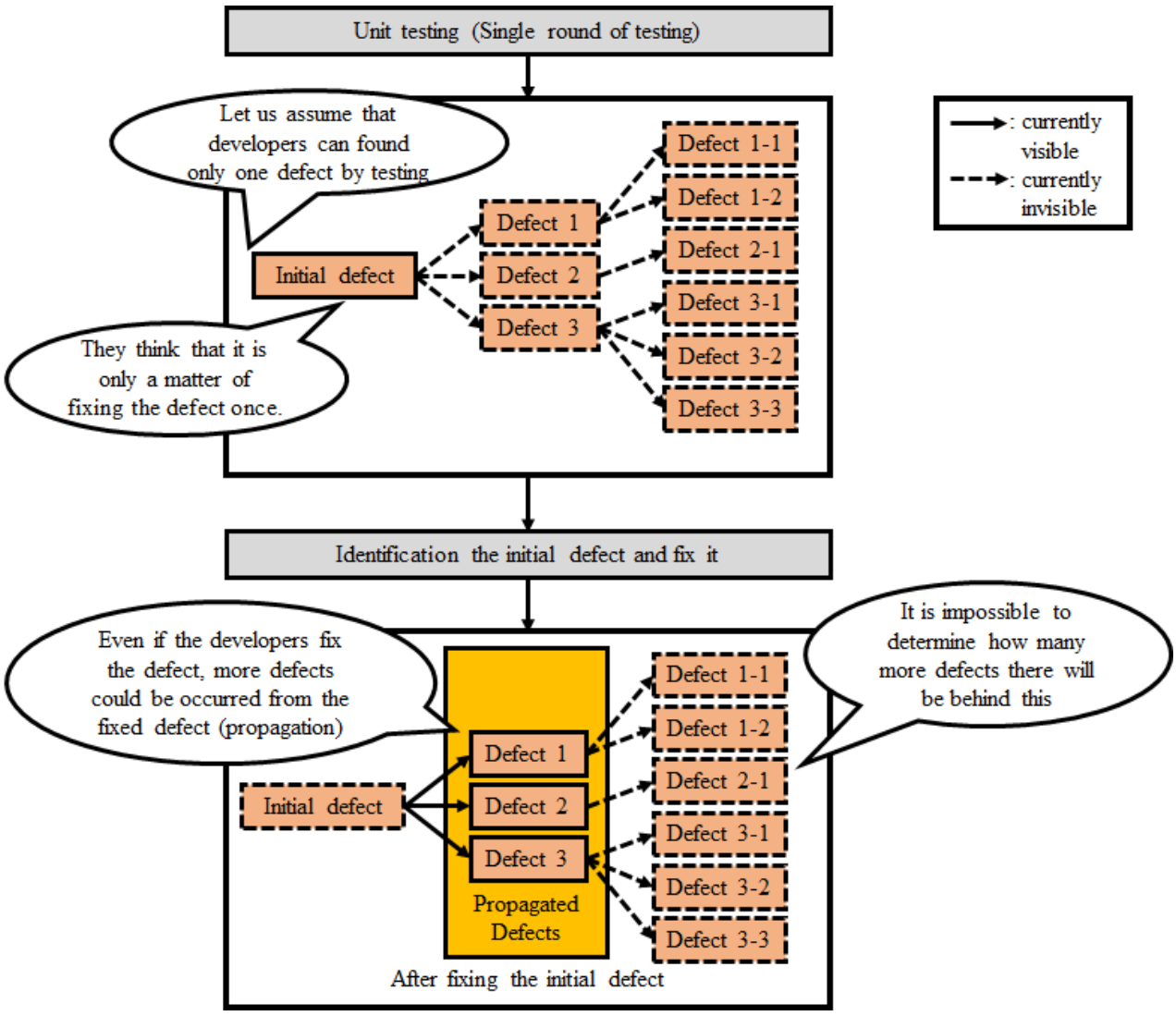

Figure 2. Occurrence of propagated defects.

In such a scenario, the defects that are added to codes B and C influence each other. More specifically, the defect in code B hides the one in code C. From the developer's perspective, the test results do not contain any problem; therefore, a code containing two flaws can be passed on as perfect

code. However, because the code passes this test case, it may cause significant problems in the future. Even if the test suite is strengthened through mutation testing or other methods, and the defect in code C can be discovered and fixed by other test cases added to the strengthened process, the defect in code B will continue to exist, and the program will remain imperfect. Therefore, the developer will need to conduct new tests, which increases the cost.

In this study, we define the connectivity of related defects to "propagated defects". To fix propagated defects, developers are required to repeat testing. However, testing over a single round may not be sufficient because it is impossible to determine the number of post-propagated defects. Thus, the cost of fixing propagated defects cannot be determined in advance, and if the propagated defects are identified without proper planning, astronomical costs are incurred. Figure 2 shows the process of occurrence of propagated defects. In the figure, the solid lines and dotted lines represent the propagated defects that can and cannot be identified with a single round of testing, respectively. In the case of an initial defect, developers can only address such a defect only through testing. As a general procedure of software testing and maintenance, developers perform testing to fix the defects. Then, in the status after the defect fix, the problem happened. However, if only the initial defect is fixed, its influences are propagated to other source codes including functions, libraries, and files. As a result, more defects occur, and their influences are then propagated to other source codes.

Unfortunately, recent studies on software testing have focused on single rounds of testing [2,23] and the defects that arise as a result of it [30,31]. Although we believe that the findings of these studies will help improve the efficiency of defect identification and fixing, they do not facilitate the identification of propagated defects, but rather only highlight propagated defects because of the high accuracy of initial defect identification. In short, simply improving the performance of the test cases does not guarantee that all the defects can be identified via initial testing alone. However, from another perspective, fixing defects that cannot be identified through initial testing may help reduce the number of test rounds. If a method to identify particular defects is available, it may become possible to identify propagated defects through initial testing alone; this will play a key role in reducing testing costs. That is, once an initial defect occurs, huge costs are incurred, which hampers the entire project [32,33]. Because the propagated defects often cause software project failure and significant cost to fix all the defects, academic researchers and practitioners are studying to address critical issues [34,35].

In this paper, a new method is proposed for identifying these propagated defects. The proposed method uses mutants that are applied in mutation testing to identify defects that are not discovered in a single round of testing. Two or more mutants are applied to different lines of code, and by doing so, the relationships between such lines are defined and recorded. As a result, these relationships are found using defects that are discovered through a single round of testing, and possible defects are suggested from among the recorded candidates. To evaluate the proposed method, Defects4J, which has been widely applied in the field of defect prediction, is used in this study to evaluate the extent to which the proposed method actually identifies propagated defects.

The contributions of this paper are as follows.

- A method is proposed to reduce the cost of software testing by identifying propagated defects.
- Data used in an actual project are applied to evaluate whether the proposed method is better than other conventional methods.

The remainder of this paper is as follows. Section 2 examines related studies, and Section 3 explores how the proposed method actually identifies propagated defects. Section 4 presents the experiments on the proposed method. Finally, Section 5 provides some concluding remarks as well as

areas of future research.

2. Related work

2.1. Mutation testing

In general, in software engineering, mutation testing refers to a method of using mutants to evaluate the appropriateness of the test cases of a test suite [36,37]. This is a fault-based testing method that modifies a portion of the program code to deliberately induce errors. As shown in Figure 3, through this process, the existing program and a program that is slightly different from the original are created, and both the original and modified programs are run through prepared test cases to determine whether they can be distinguished (identifying errors that were inserted to create the modified program) using the current test suite [38–41]. If the execution results of the modified program differ from those of the original program, it indicates that the modified program can be distinguished by the current test suite. In this case, the modified program is treated as "killed" and is no longer of interest to the developer. By contrast, if the execution results of the modified program are the same as those of the original program, it means that the modified program cannot be distinguished by the current test suite. In this case, the modified program is called a "live mutant," and is proof that the test cases of the current test suite do not access all areas of the deliberately changed code. In other words, the existence of a live mutant means that the test suite was insufficient to kill the modified program. In mutation testing, the goal of the developer is to add new test cases to the test suite such that all of the live mutants that were identified by the testing results can be killed [42–45].

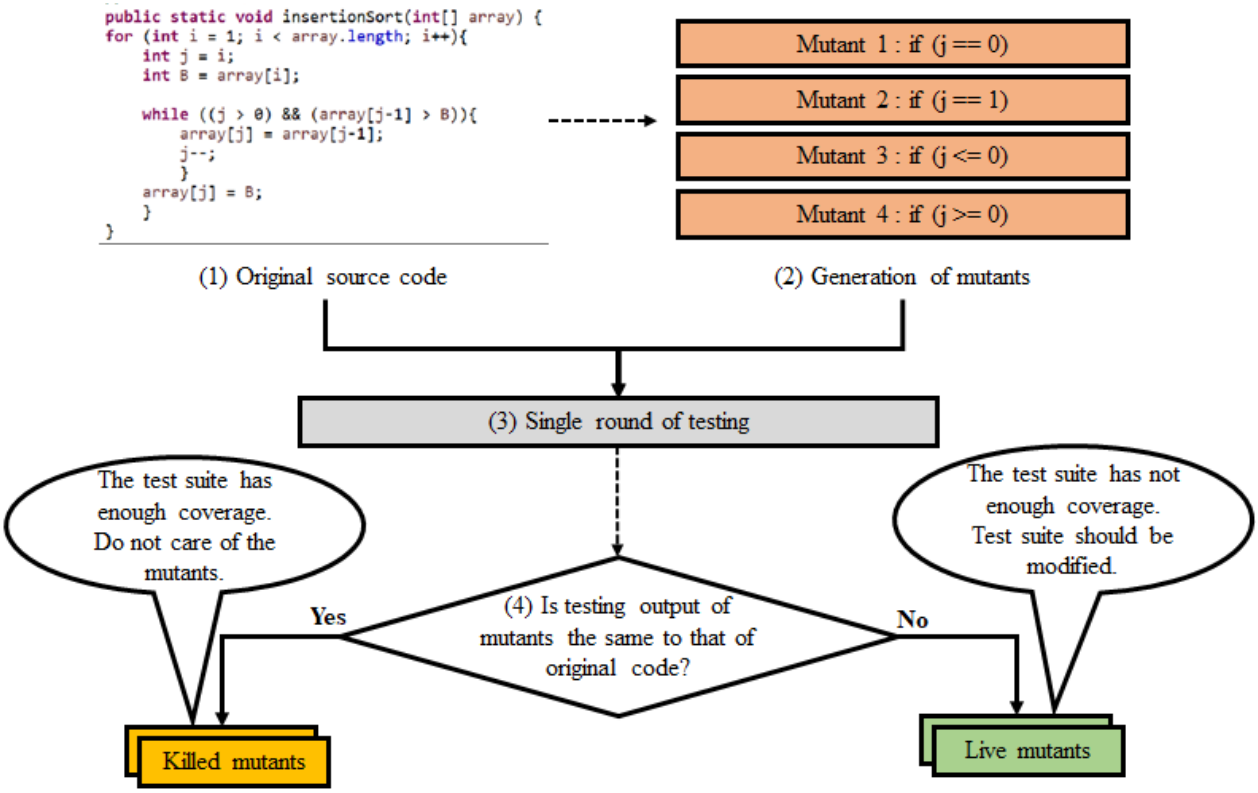

Figure 3. General process of Mutation testing.

Research on mutation testing is being actively conducted. In particular, studies [46,47] have used Higher Order Mutants (HOMs), created by combining two or more mutants, rather than mutants that are used in normal mutation tests. Although mutation testing that applies HOMs yields robust results, it is difficult for HOMs themselves to survive, and studies are being conducted to improve on this. As mentioned before, the goal of mutation testing is basically to improve the quality of the test cases. Therefore, in related studies, mutation testing has been used in combination with test cases, and it is difficult to find studies that have only used mutation testing to identify defects [45]. In this paper, a method is proposed that uses nested mutation testing to find propagated defects, which are generally difficult to find during the initial round of testing.

2.2. Software defect prediction

Software defect prediction (SDP) refers to predicting whether defects exist in a program before actually confirming the existence of defects through testing or maintenance [30,48]. The simplest code-based SDP method collects examples of defects occurring in the code of several projects (buggy code) and then finds code that is similar to the buggy code [49–51]. However, this incurs a problem in which only patterns that are similar to the collected defects can be found, and even when defects are collected, if the quantity is small compared to categories with a large number of data, the classifier may not cover such defects properly [52]. The most common method used in SDP is a method based on learning models that treat the project code as features [53,54]. By learning the code, the method learns for itself the meaning and methods by which the code was written and finds any faulty or updated code that no longer needs to be used. The learning performance of model-based methods is extremely dependent on the data used as features; therefore, there are often cases in which efforts are made to preprocess the data through methods such as normalization or resampling.

In defect document based SDP, text-based analysis and learning are common [55]. The main issue in defect documents is the automation of the defect document classification process based on severity and priority, which is known as triage. Various benefits can be obtained by applying such triage; for example, developers can deal with higher-priority and fatal defects first, and duplicated defect documents or documents that are not defects can be processed in advance to improve the task efficiency. Therefore, triage is an extremely important task in software engineering. Triage is normally applied manually. There have recently been cases in which image and video data are included according to the software; in general, however, most defect documents consist of natural language and code, and therefore triage often uses Natural Language Processing (NLP) and Recurrent Neural Network (RNN)-type deep learning methods, which have advantages in regard to text learning [56–58].

Fault localization makes up the main part of SDP. The most commonly used type of fault localization is spectrum-based [29,31], and fault localization is applied using a program spectrum that illustrates which parts of a program are active. Fault localization methods are called Ochiai method [60–63] or Jaccard method [64–67]. The methods according to the correlation coefficient used.

3. Approach

This section explains how the proposed method can be used to identify propagated defects. Figure 4 shows the overall process of the proposed method

3.1. Analyzing defects with dual mutation testing

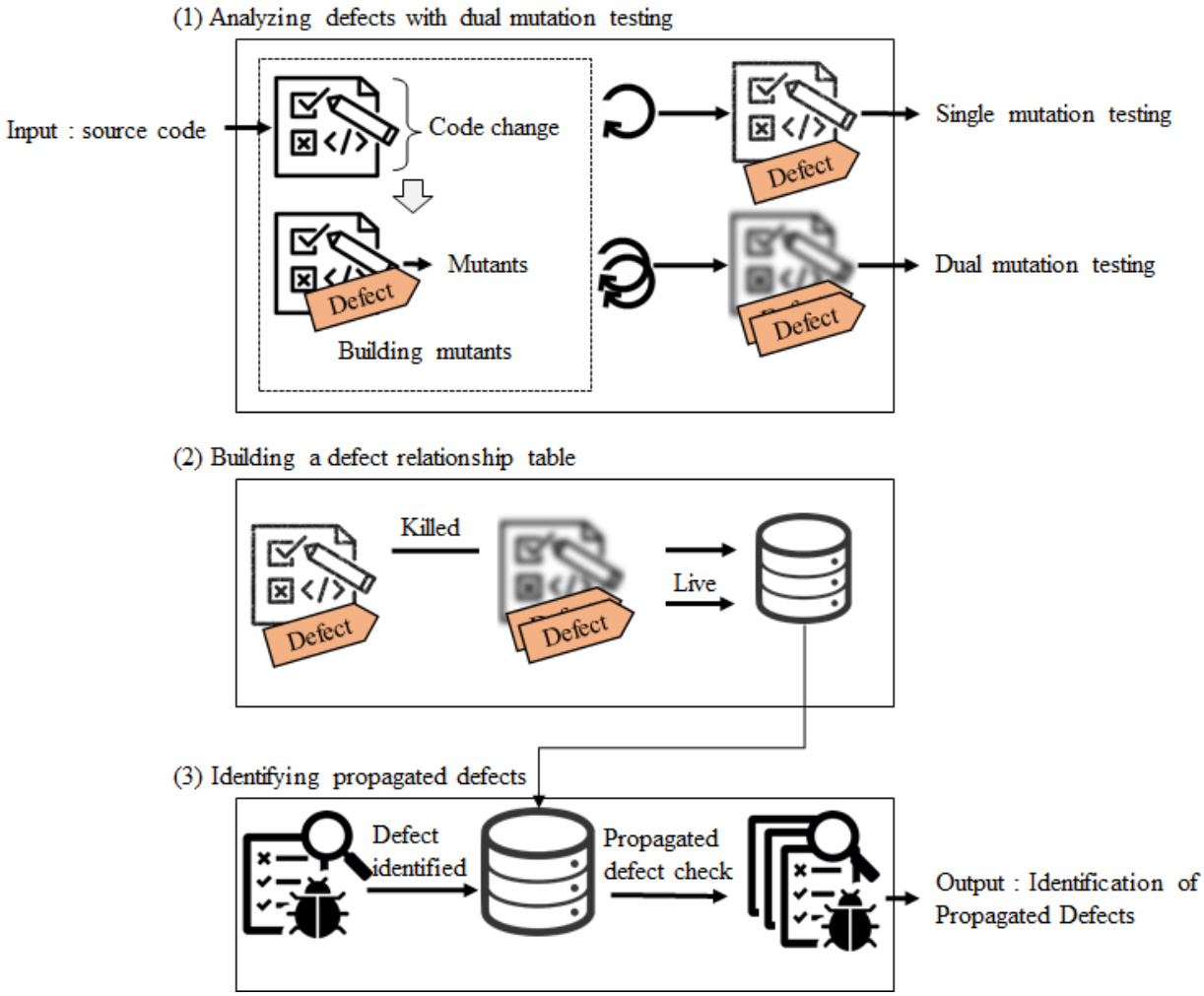

Figure 4. Overall approach of the proposed method for identifying propagated defects.

The proposed method receives the source code and test suite of the original target program as input. First, mutation testing was performed through a set of mutation operations applied to the existing source code with the prepared test suite (the first-round mutation testing which is called single mutation testing in this study), and then the mutation testing was conducted again to the generated mutants (the second-round mutation testing which is called as dual mutation testing in this study). That is, the proposed method uses two types of mutations: in which a mutation operation is applied from the single mutation testing, in which a mutation operation is applied one more time on mutants generated from the single mutation testing. In order to perform the above testing process, a single defect is deliberately inserted into the original source code, and the modified source code is then evaluated based on the prepared test case. Another single defect is then deliberately inserted into the source code that was just modified, and the modified source code is evaluated using the prepared test case (from the perspective of the original source code, two different defects are inserted). In the results, the mutation is saved as killed if it fails to pass even one test case, and is saved as live if it passes all test cases.

3.2. Building a defect relationship table

If the single mutation test result is killed and the dual mutation test result is live, the single and dual mutation results (including the modification information), as well as the test case in which the single mutation is killed, are all saved. If the single mutation is killed, it means the defect caused by the mutant can be identified by the current test suite, and if the dual mutation is live, it means that the defect caused by the mutant cannot be identified by the current test suite. Therefore, a defect found in a single mutant state is influenced by the dual mutant and obscured, and thus cannot be identified by the current test suite. As such, propagated defects can be found by creating a defect relationship table by saving the modified code for creating each mutant as well as the test cases by which the defects were discovered.

3.3. Identifying propagated defects

After the defect relationship table is completed, if there are test cases that cannot be passed in the test results of the software during the program operation, the developer finds the reason why the test case failed and fixes the defect that fails during the testing. As a result, the code is changed during the process of fixing the defects. However, because hidden defects may still exist in the code, the developer can explore whether additional propagated defects exist by searching for test cases related to the fixed defects in the defect relationship table or searching for changed lines of code in the defect relationship table. This process also saves killed single mutation test cases; therefore, the test costs can be reduced by applying only the saved test cases without the need for complete testing. If the test cases fail and it is discovered that a defect exists in the program, the developer can fix the defect at a low cost, and if there are no defects, the test cases will succeed, which does not incur significant problems in regard to the overall test costs.

4. Evaluation

4.1. Experimental design

The Defects4J dataset was used to evaluate the proposed method [68]. Defects4J provides 835 defects from 17 actual projects using Java. Defects4J provides not only the defect information of the project but also documents and code regarding the version containing the defect, allowing it to be known how the defect and the defect document have affected each other. In addition, it also contains a test suite for the testing itself, as well as the compiled and testing functions such that the same results are guaranteed in whatever environment the testing is run. Table 1 shows a brief summary of the projects and defects that are provided by Defects4J (version 2.0.0).

Table 1. Summary of Defects4J (version 2.0.0).

Identifier	Project name	Number of defects	Active defect IDs	Deprecated defect IDs
Chart	jfreechart	26	1–26	None
Cli	commons-cli	39	1–5, 7–49	6
Closure	closure-compiler	174	1–62, 64–92, 94–176	63, 93
Codec	commons-codec	18	1–18	None
Collections	commons-collections	4	25–28	1-24
Compress	commons-compress	47	1–47	None
Csv	commons-csv	16	1–16	None
Gson	gson	18	1–18	None
JacksonCore	jackson-core	26	1–26	None
JacksonDatabind	jackson-databind	112	1–112	None
JacksonXML	jackson-dataformat-xml	6	1–6	None
Jsoup	jsoup	93	1–93	None
JxPath	commons-jxpath	22	1–22	None
Lang	commons-lang	64	1, 3–65	2
Math	commons-math	106	1–106	None
Mockito	mockito	38	1–38	None
Time	joda-time	26	1–20, 22–27	21

* A description of each column
- Identifier: acronym to distinguish projects in Defects4J.
- Project name: actual project name searched in Java library.
- Number of defects: the number of defects provided by Defects4J.
- Active defect IDs: ID numbers for defects used in Defects4J
- Deprecated defect IDs: ID number for defects no longer used in Defects4J.

Among the Defects4J projects, JacksonXML is used in our experiment, because the enough number of propagated defects exist. In general, the projects with the smaller number of defects (e.g., Collections) cannot build any propagated defects, while the projects with the greater number of defects (e.g., Lang, Time) spend too much time on buildng propagated defects. Our experiments were conducted by applying the proposed method to the original code of JacksonXML. Although Defects4J

does not directly provide mutation testing functions, because it is limited to single mutations, single mutation operators were extracted, and dual mutations were then applied. The modified code of each mutant and the test results were recorded when the single mutation result was killed and the dual mutation result was live. Next, pairs of defects were prepared from the six defects of JacksonXML to create 30 ordered pairs. Supposing that the defect ordered pairs were expressed in the form of (a, b), defect a was first applied to the original code, and defect b was then additionally applied. The 30 propagated defects that were obtained as a result were used as targets for identification based on the recorded nested mutation results. In order to evaluate the proposed method, the existing seven fault localization methods shown in Table 2 are applied to determine whether the methods could identify the 30 propagated defects.

Table 2. Seven fault localization methods used in our experiments.

Fault localization methods	Description of the core formulas
Tarantula [60,61,64]	$\frac{\frac{failed(s)}{totalfailed}}{\frac{failed(s)}{totalfailed}+\frac{passed(s)}{totalpassed}}$
Ochiai [60,64]	$\frac{failed(s)}{\sqrt{totalfailed \cdot (failed(s)+passed(s))}}$
Opt2 [64]	$failed(s) - \frac{passed(s)}{totalpassed+1}$
Barinel [61,64]	$1 - \frac{passed(s)}{passed(s)+failed(s)}$
Dstar2 [60,61,64]	$\frac{failed(s)^2}{passed(s)+(totalfailed-failed(s))}$
Muse [64]	$\underset{m \in mut(s)}{avg} \left[failed(m) - \frac{totalfailed}{totalpassed} passed(m)\right]$
Jaccard [60,64]	$\frac{failed(s)}{totalfailed+passed(s)}$

* A description of each column
- Fault localization methods: the name of methods used in the experiments.
- Description of the core formula: formulas for applying the fault localization methods

* A description of each argument in the formulas
- totalfailed: the total number of tests that failed.
- totalpassed: the total number of tests that passed.
- failed(s): the number of tests that failed for the statement s.
- passed(s): the number of tests that passed for the statement s.
- mut(s): a set of mutants for s.
- failed(m): the number of tests that failed for the mutant m.
- passed(m): the number of tests that passed for the mutant m.

Table 2 shows a summary of fault localization methods. The range of the core formula applied

for the methods is from 0 to 1, and the result of this formula is called "suspiciousness". The closer the suspiciousness is to 1, the more similar the source code line is to a defect. There is no a clear value that a source code line is a defect. In this paper, we use that the criteria for determination of propagated defects is average suspiciousness of the defects on current source codes. That is, we do not allow them as propagated defects unless the suspiciousness of propagated defects exceeds suspiciousness of the defects on current source codes.

4.2. Experimental results

The experimental results showed that 5 out of the 30 propagated defects could be identified in Table 3. Table 3 provides a simple summary of the propagated defects that the proposed method was able to identify. The proposed method could discern that defects 2 and 3 affected the other defects twice and were therefore obscured. In addition, it was seen that defects 5 and 6 did not affect the other defects when the proposed method was used.

Table 3. Propagated defects identified using the proposed method.

Defects Pair	Single mutants file	Single mutant line number	Dual mutant file	Dual mutant line number
1-2	FromXmlParser.java	321	FromXmlParser.java	506
1-3	FromXmlParser.java	321	XmlTokenStream.java	322
2-3	FromXmlParser.java	563	XmlTokenStream.java	315
3-4	FromXmlParser.java	713	FromXmlParser.java	660
4-2	FromXmlParser.java	656	FromXmlParser.java	427

* A description of each column
- Defects Pair: Active defect ID pair of propagated defects identified by the proposed method.
- Single mutants file: a file name with the single mutation applied.
- Single mutant line number: a line number modified with the single mutation applied.
- Dual mutants file: a file name with the dual mutation applied.
- Dual mutant line number: a line number modified with the dual mutation applied.

As presented in Table 4, the existing fault localization methods were unable to identify most of the propagated defects. Only Tarantula, Opt2, and Muse were able to identify propagated defects (1, 2), and the other methods were unable to identify any of the given propagated defects. The existing methods could identify defects in current original source code. However, they did not show the enough performance to identify the propagated defects because fault localization methods require testing results. That is, if developers cannot test all source codes related to all the propagated defects, they cannot predict where the propagated defects will occur. This issue is the critical limitation of the existing methods and is why the existing methods are not suitable for the identification of propagated defects.

Table 4. Propagated defects identified through existing fault localization.

Fault localization methods	Defects pair
Tarantula [60,61,64]	1-2
Ochiai [60,64]	-
Opt2 [64]	1-2
Barinel [61,64]	-
Dstar2 [60,61,64]	-
Muse [64]	1-2. 4-2
Jaccard [60,64]	-

* A description of each column
- Fault localization methods: the name of methods used in the experiments.
- Defects pair: Active defect ID pairs of propagated defects identified by fault localization methods.

5. Conclusions

In software engineering, testing is an area of software maintenance that has been studied for a long time. However, testing is extremely expensive, and it cannot be guaranteed that all defects will be found during a single round of testing. Therefore, the resolution of defects that are not discovered by a single round of testing is important for reducing the test costs. In this paper, an approach is proposed that uses the mutants in a mutation test to identify defects that are not discovered by a single round of testing. The proposed method simultaneously applies two or more mutants to a program to define and record the relationships between different lines of code. In turn, these relationships are examined using the defects that were discovered by a single round of testing, and possible defects are recommended from among the recorded candidates. To evaluate the proposed method, a comparative study was conducted using the fault localization method, which is commonly employed in defect prediction, as well as the Defects4J defect prediction dataset, which provides code and defects extracted from actual projects and is widely used in software defect prediction. The results of the evaluation showed that the proposed method achieves a better performance than the seven fault localization approaches. In addition to applying the proposed method for defect detection, in future studies, the relationships between codes will be considered to determine the conversion sequences.

Acknowledgments

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2020R1I1A3073313).

Conflict of interest

The authors declare that they have no conflicts of interest to report regarding the present study.

References

1. R. H. Ur, R. Mushtaq, A. Palwasha, K. Mukhtaj, I. Nadeem, K. H. Ullah, Making the sourcing decision of software maintenance and information technology, *IEEE Access*, **9** (2021), 11492–11510. https://doi.org/10.1109/ACCESS.2021.3051023
2. F.S. Ana M, M. R. Chaudron, M. Genero, An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles, *Empirical Software Eng.*, **23** (2018), 3281–3345. https://doi.org/10.1007/s10664-018-9599-4
3. E. Vahid, O. Bushehrian, G. Robles, Task assignment to counter the effect of developer turnover in software maintenance: A knowledge diffusion model, *Inf. Software Technol.*, **143** (2022), 106786. https://doi.org/10.1016/j.infsof.2021.106786
4. K. Jang, W. Kim, A method of activity-based software maintenance cost estimation for package software, *J. Supercomput.*, **78** (2021), 8151–8171. https://doi.org/10.1007/s11227-020-03610-6
5. T. Masateru, M. Akito, M. Kenichi, O. Sawako, O. Tomoki, Analysis of work efficiency and quality of software maintenance using cross-company dataset, *IEICE Trans. Inf. Syst.*, **104** (2021), 76–90. https://doi.org/10.1587/transinf.2020MPP0004
6. K.W. Kim, Y. Son, Software weakness evaluation method for secure software development, in *Proceedings on 2021 International Conferences on Multimedia Information Technology and Applications*, (2021), 322–325.
7. C. Kim, D. Kim, H. Kang, Detecting defect in headlamp housing with machine learning techniques, in *Proceedings on 2021 International Conferences on Multimedia Information Technology and Applications*, (2021), 428–430.
8. Y. J. Choi, Y. W. Lee, B. G Kim, Residual-based graph convolutional network for emotion recognition in conversation for smart Internet of Things, *Big Data*, **9** (2021), 279–288. https://doi.org/10.1089/big.2020.0274
9. P. P. Roy, P. Kumar, B.G. Kim, An efficient sign language recognition (SLR) system using Camshift tracker and hidden Markov model (hmm), *SN Comput. Sci.*, **2** (2021), 1–15. https://doi.org/10.1007/s42979-021-00485-z
10. B. George, F. Stefan, M. Michael, P. Josef, An early investigation of unit testing practices of component-based software systems, in *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests*, (2020), 12–15. https://doi.org/10.1109/VST50071.2020.9051632
11. M. Alcon, H. Tabani, J. Abella, F. J. Cazorla, Enabling Unit Testing of Already-Integrated AI Software Systems: The Case of Apollo for Autonomous Driving, in *2021 24th Euromicro Conference on Digital System Design,* (2021), 426–433. https://doi.org/10.1109/DSD53832.2021.00071
12. D. Xavier, S. Panichella, A. Gambi, Java unit testing tool competition: Eighth round, in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, (2020), 545–548. https://doi.org/10.1145/3387940.3392265
13. T. Mengqing, J. Yan, W. Xiang, P. Rushu, Black-box approach for software testing based on fat-property, in *MATEC Web of Conferences,* **309** (2020), 02008. https://doi.org/10.1051/matecconf/202030902008
14. S. Bo, Y. Shao, C. Chen, Study on the automated unit testing solution on the linux platform, in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion*, (2019), 358–361. https://doi.org/10.1109/QRS-C.2019.00073

15. M. Héctor D, J. Gunel, S. Federica, T. Paolo, C. David, Diversifying focused testing for unit testing, *ACM Trans. Software Eng. Method.*, **30** (2021), 1–24. https://doi.org/10.1145/3447265
16. X. Wang, L. Wei, B. Tao, S. Ji, A study about unit testing for embedded software of control system in nuclear power plant, in *International Symposium on Software Reliability, Industrial Safety, Cyber Security and Physical Protection for Nuclear Power Plant*, (2020), 157–163. https://doi.org/10.1007/978-981-16-3456-7_17
17. F. Anfal A, R. G. Alsarraj, A. M. Altaie, Software cost estimation based on dolphin algorithm, *IEEE Access*, **8** (2020), 75279–75287. https://doi.org/10.1109/ACCESS.2020.2988867
18. C. Sonia, H. Singh, Optimizing design of fuzzy model for software cost estimation using particle swarm optimization algorithm, *Int. J. Comput. Intell. Appl.*, **19** (2020), 2050005. https://doi.org/10.1142/S1469026820500054
19. A. Farrukh, A review of machine learning models for software cost estimation, *Rev. Comput. Eng. Res.*, **6** (2019), 64–75. https://doi.org/10.18488/journal.76.2019.62.64.75
20. K. Ishleen, N. G. Singh, W. Ritika, J. Vishal, B. Anupam, Neuro fuzzy—COCOMO II model for software cost estimation, *Int. J. Inf. Technol.*, **10** (2018), 181–187. https://doi.org/10.1007/s41870-018-0083-6
21. J. Miller, S. Wienke, M. Schlottke-Lakemper, M. Meinke, M. S. Müller, Applicability of the software cost model COCOMO II to HPC projects, *Int. J. Comput. Sci. Eng.*, **17** (2018), 283–296. https://doi.org/10.1504/IJCSE.2018.095849
22. A. Asheeri, M. Mohd, M. Hammad, Machine learning models for software cost estimation, in *2019 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies*, (2019), 1–6. https://doi.org/10.1109/3ICT.2019.8910327
23. A. Shaina, N. Mishra, Software cost estimation using artificial neural network, in *Soft Computing: Theories and Applications,* Springer, (2018), 51–58. https://doi.org/10.1007/978-981-10-5699-4_6
24. S. S. Pratap, V. P. Singh, A. K. Mehta, Differential evolution using homeostasis adaption based mutation operator and its application for software cost estimation, *J. King Saud Univ.-Comput. Inf. Sci.*, **33** (2021), 740–752. https://doi.org/10.1016/j.jksuci.2018.05.009
25. S. W. Ahmad, G. R. Bamnote, Whale-crow optimization (WCO)-based optimal regression model for software cost estimation, *Sādhanā*, **44** (2019), 1–15. https://doi.org/10.1007/s12046-019-1085-1
26. J. A. Khan, S. U. R. Khan, J. Iqbal, I. U. Rehman, Empirical investigation about the factors affecting the cost estimation in global software development context, *IEEE Access*, **9** (2021), 22274–22294. https://doi.org/10.1109/ACCESS.2021.3055858
27. V. S. Desai, R. Mohanty, ANN-Cuckoo optimization technique to predict software cost estimation, in *2018 Conference on Information and Communication Technology*, (2018), 1–6. https://doi.org/10.1109/INFOCOMTECH.2018.8722380
28. R. C. A. Alves, D. A. G. Oliveira, G. A. N. Segura, C. B. Margi, The cost of software-defining things: A scalability study of software-defined sensor networks, *IEEE Access* **7** (2019), 115093–115108. https://doi.org/10.1109/ACCESS.2019.2936127
29. D. G. Lee, Y. S. Seo, Testing cost reduction using nested mutation testing, in *Proceedings on 2021 International Conferences on Multimedia Information Technology and Applications*, (2021), 462–464.

30. N. Li, M. Shepperd, Y. Guo, A systematic review of unsupervised learning techniques for software defect prediction, *Inf. Software Technol.*, **122** (2020), 106287. https://doi.org/10.1016/j.infsof.2020.106287
31. F. Keller, L. Grunske, S. Heiden, A. Filieri, A. Hoorn, D. Lo, A critical evaluation of spectrum-based fault localization techniques on a large-scale software system, in *2017 IEEE International Conference on Software Quality, Reliability and Security*, (2017), 114–125. https://doi.org/10.1109/QRS.2017.22
32. *The cost of poor software quality in the US: A 2020 report*, Report of Proc. Consortium Inf. Softw. QualityTM, 2021. Available from: https://www.disputesoft.com/wp-content/uploads/2021/01/CPSQ-2020-Software-Report.pdf.
33. 'Fully weaponized' software bug poses a threat to Minecraft gamers and apps worldwide including Google, Twitter, Netflix, Spotify, Apple iCloud, Uber and Amazon, 2021. Available from: https://www.dailymail.co.uk/news/article-10297693/Global-race-patch-critical-computer-bug.html.
34. P. Vitharana, Defect propagation at the project-level: results and a post-hoc analysis on inspection efficiency, *Empirical Software Eng.*, **22** (2017), 57–79. https://doi.org/10.1007/s10664-015-9415-3
35. Z. Wei, T. Shen, X. Chen, Just-in-time defect prediction technology based on interpretability technology, in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, (2021), 78–89. https://doi.org/10.1109/DSA52907.2021.00017
36. L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. J. Xu, et al., Deepmutation: Mutation testing of deep learning systems, in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, (2018), 100–111. https://doi.org/10.1109/ISSRE.2018.00021
37. G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, R. Just, An industrial application of mutation testing: Lessons, challenges, and research directions, in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, (2018), 47–53. https://doi.org/10.1109/ICSTW.2018.00027
38. Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, J. Zhao, DeepMutation++: A mutation testing framework for deep learning systems, in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (2019), 1158–1611. https://doi.org/10.1109/ASE.2019.00126
39. P. Gómez-Abajo, E. Guerra, J. D. Lara, M. G. Merayo, Wodel-Test: a model-based framework for language-independent mutation testing, *Software Syst. Model.*, **20** (2021), 767–793. https://doi.org/10.1007/s10270-020-00827-0
40. L. Chen, L. Zhang, Speeding up mutation testing via regression test selection: An extensive study, in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, (2018), 58–69. https://doi.org/10.1109/ICST.2018.00016
41. N. Humbatova, G. Jahangirova, P. Tonella, DeepCrime: mutation testing of deep learning systems based on real faults, in *Proceedings of the 30th ACM SIGSOFT International, Symposium on Software Testing and Analysis*, (2021), 67–78. https://doi.org/10.1145/3460319.3464825
42. K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, et al., MDroid+: A mutation testing framework for android, in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, (2018), 33–36. https://doi.org/10.1145/3183440.3183492

43. Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, Z. Chen, MuSC: A tool for mutation testing of ethereum smart contract, in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (2019), 1198–1201. https://doi.org/10.1109/ASE.2019.00136
44. D. Mao, L. Chen, L. Zhang, An extensive study on cross-project predictive mutation testing, in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, (2019), 160–171. https://doi.org/10.1109/ICST.2019.00025
45. D. Cheng, C. Cao, C. Xu, X. Ma, Manifesting bugs in machine learning code: An explorative study with mutation testing, in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, (2018), 313–324. https://doi.org/10.1109/QRS.2018.00044
46. S. Lee, D. Binkley, R. Feldt, N. Gold, S. Yoo, Causal program dependence analysis, preprint, arXiv:2104.09107. https://doi.org/10.48550/arXiv.2104.09107
47. S. Oh, S. Lee, S. Yoo, Effectively sampling higher order mutants using causal effect, in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops*, (2021), 19–24. https://doi.org/10.1109/ICSTW52544.2021.00017
48. X. Cai, Y. Niu, S. Geng, J. Zhang, Z. Cui, J. Li, et al., An under-sampled software defect prediction method based on hybrid multi-objective cuckoo search, *Concurrency Comput.: Pract. Exper.*, **32** (2020), e5478. https://doi.org/10.1002/cpe.5478
49. A. Rahman, J. Stallings, L. Williams, Defect prediction metrics for infrastructure as code scripts in DevOps, in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, (2018), 414–415. https://doi.org/10.1145/3183440.3195034
50. A. Amar, P. C. Rigby, Mining historical test logs to predict bugs and localize faults in the test logs, in *2019 IEEE/ACM 41st International Conference on Software Engineering*. (2019), 140–151. https://doi.org/10.1109/ICSE.2019.00031
51. P. Sangameshwar, B. Ravindran, Predicting software defect type using concept-based classification, *Empirical Software Eng.*, **25** (2020), 1341–1378. https://doi.org/10.1142/S1469026820500054
52. S. Wang, T. Liu, J. Nam, L. Tan, Deep semantic feature learning for software defect prediction, *IEEE Trans. Software Eng.*, **46** (2018), 1267–1293. https://doi.org/10.1109/TSE.2018.2877612
53. X. Yin, L. Liu, H. Liu, Q. Wu, Heterogeneous cross-project defect prediction with multiple source projects based on transfer learning, *Math. Biosci. Eng.*, **17** (2020), 1020–1040. https://doi.org/10.3934/mbe.2020054
54. L. Qiao, X. Li, Q. Umer, P. Guo, Deep learning based software defect prediction, *Neurocomputing*, **385** (2020), 100–110. https://doi.org/10.1016/j.neucom.2019.11.067
55. F. Wu, X. Y. Jing, Y. Sun, J. Sun, L. Huang, F. Cui, et al., Cross-project and within-project semisupervised software defect prediction: A unified approach, *IEEE Trans. Reliab.*, **67** (2018), 581–597. https://doi.org/10.1109/TR.2018.2804922
56. D. L. Miholca, G. Czibula, I. G. Czibula, A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks, *Inf. Sci.*, **441** (2018), 152–170. https://doi.org/10.1016/j.ins.2018.02.027
57. A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, H. Haghighi, SLDeep: Statement-level software defect prediction using deep-learning model on static code features, *Expert Syst. Appl.*, **147** (2020), 113156. https://doi.org/10.1016/j.eswa.2019.113156

58. G. G. Cabral, L. L. Minku, E. Shihab, S. Mujahid, Class imbalance evolution and verification latency in just-in-time software defect prediction, in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, (2019), 666–676. https://doi.org/10.1109/ICSE.2019.00076
59. A. Perez, R. Abreu, A. Deursen, A test-suite diagnosability metric for spectrum-based fault localization approaches, in *2017 IEEE/ACM 39th International Conference on Software Engineering*, (2017), 654–664. https://doi.org/10.1109/ICSE.2017.66
60. X. Li, W. Li, Y. Zhang, L. Zhang, Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (2019), 169–180. https://doi.org/10.1145/3293882.3330574
61. A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, R. A. Rasheed, Multiple fault localization of software programs: A systematic literature review, *Inf. Software Technol.*, **124** (2020), 106312. https://doi.org/10.1016/j.infsof.2020.106312
62. Z. Li, Y. Wu, Y. Liu, An empirical study of bug isolation on the effectiveness of multiple fault localization, in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, (2019), 18–25. https://doi.org/10.1109/QRS.2019.00016
63. H. L. Ribeiro, R. P. A. de Araujo, M. L. Chaim, H. A. de Souza, F. Kon, Jaguar: A spectrum-based fault localization tool for real-world software, in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, (2018), 404–409. https://doi.org/10.1109/ICST.2018.00048
64. K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, Y. Le Traon, You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems, in *2019 12th IEEE conference on software testing, validation and verification (ICST)*, (2019), 102–113. https://doi.org/10.1109/ICST.2019.00020
65. D. Zou, J. Liang, Y. Xiong, M. D. Ernst, L. Zhang, An empirical study of fault localization families and their combinations, *IEEE Trans. Software Eng.*, **47** (2019), 332–347. https://doi.org/10.1109/TSE.2019.2892102
66. J. Kim, J. Kim, E. Lee, Variable-based fault localization, *Inf. Software Technol.*, **107** (2019), 179–191. https://doi.org/10.1016/j.infsof.2018.11.009
67. Y. Kim, S. Mun, S. Yoo, M. Kim, Precise learn-to-rank fault localization using dynamic and static features of target programs, *ACM Trans. Software Eng. Method. (TOSEM)*, **28** (2019), 1–34. https://doi.org/10.1145/3345628
68. *Defects4J*, 2022. available from: https://github.com/rjust/defects4j.

AIMS Press