*Journal of Computer Science and Cybernetics, V.40, N.2 (2024), 165–185*
DOI no. 10.15625/1813-9663/19694

# CLUE: A CLUSTERING-BASED TEST REDUCTION APPROACH FOR SOFTWARE PRODUCT LINES

HIEU DINH VO*, THU-TRANG NGUYEN

*Faculty of Information Technology, VNU University of Engineering and Technology, Ha Noi, Viet Nam*

**Abstract.** Nowadays, organizations have increasingly turned to software product lines (SPLs)/ configurable systems to meet diverse user needs and market demands. SPLs offer configurability and scalability but pose a formidable challenge in testing. Assuring the quality of variants across numerous configurations demands innovative testing strategies that balance effectiveness and efficiency. To improve testing productivity, test reduction techniques have been widely employed in non-configurable code. However, test reduction for SPL systems remains mostly unexplored.

This paper introduces Clue, a novel test reduction approach to enhance testing productivity in SPLs. Our idea is that to effectively and efficiently reveal failures, tests should be distinctive and cover diverse behaviors of the system. Meanwhile, similar tests covering the same/similar behaviors of an SPL system do not need to be redundantly executed multiple times. Clue clusters the system's tests into distinctive groups containing similar tests. Within each cluster, tests are prioritized based on the number of feature interactions, a common root cause of defects in SPLs. Clue continually selects and executes the top-prioritized test in each cluster until a bug is revealed or the allocated effort budget is exhausted. To evaluate Clue, we conducted several experiments on a dataset of six widely-used SPL systems. The results show that Clue enables developers to detect defects earlier, requiring up to 88% less effort than existing approaches. Additionally, using only 50% of the original test suites, Clue can identify most of the bugs in the buggy SPL systems while maintaining fault localization performance.

**Keywords.** Test reduction; Software product line; clustering; Feature interaction; Fault localization.

## 1. INTRODUCTION

In the dynamic landscape of software engineering, organizations are increasingly turning to software product lines (SPL) as a strategic approach to meet diverse user needs and market demands [9]. An SPL system is a family of related software products or variants sharing common core assets while allowing for extensive configurability. This approach introduces flexibility, scalability, and cost-efficiency, making it an attractive choice for delivering tailored solutions to various domains and customer segments. However, the configurability and diversity inherent in SPLs introduce a profound challenge for SPL testing [1, 9, 20, 33, 36]. In addition, to guarantee the quality of all the variants of

*Corresponding author.
*E-mail addresses*: hieuvd@vnu.edu.vn (H.D. Vo); trang.nguyen@vnu.edu.vn (T.T. Nguyen).

© 2024 Vietnam Academy of Science & Technology

the system, the potential test cases of each SPL system grow exponentially. This poses a challenge for innovative quality assurance strategies to balance thorough testing and resource efficiency.

In practice, several test reduction techniques have been proposed to improve testing productivity in non-configurable code by systematically reducing the number of test cases while retaining the same level of code coverage and fault coverage [3, 12, 14, 17, 18, 26, 51, 52]. For SPL systems where testing is usually costly, test reduction could play an even more critical role in facilitating efficient resource allocation, enabling faster testing cycles in agile and continuous integration environments, and ultimately helping organizations meet the evolving demands of their customers.

The existing studies [6, 7, 27, 32, 36, 41] primarily concentrated on reducing the number of configurations/products to test rather than directly reducing the test sets. The $t$-wise (i.e., $k$-way) sampling algorithm [32] covers all combinations of $t$ configuration options, while pair-wise checks all pairs of configuration options. A study by Medeiros et al. [32] showed that realistic constraints among options, global analysis, header files, and build-system information influence the performance of most sampling algorithms substantially, and several algorithms are no longer feasible in practice. The most-enabled-disabled algorithm [1] checks two samples independently of the number of configuration options. When there are no constraints among configuration options, it enables all options, and then it disables all configuration options. One-(enabled/disabled) algorithm [1] enables/disables one configuration option at a time. However, while reducing the number of configurations or products to test can improve test productivity, it does not always address the challenge of high-cost testing caused by large test sets in sampled products [2, 20, 32].

In this paper, we introduce CLUE, a novel test reduction approach to improve test productivity in SPL systems. Our idea is that to efficiently and effectively test the system's behaviors, tests should be distinctive and cover diverse behaviors. Similar tests covering the same/similar behaviors of an SPL system could be unnecessary to be executed repeatedly.

Particularly, for an SPL system, CLUE clusters the set of tests into distinctive groups containing similar tests. In each cluster, the tests are prioritized according to the number of *feature interactions* [9] in the related products. The reason is that most of the bugs in SPL systems are caused by undesirable interactions of the system features [1, 7, 37]. Thus, executing the tests in the product covered more interactions could increase the likelihood of revealing the bugs earlier. After that, CLUE continuously selects and executes the top-prioritized test in each cluster that has not been executed until a bug is revealed (a test is failed) or the effort budget is reached.

We conducted several experiments to evaluate the SPL test reduction performance of CLUE on a large public dataset containing six SPL systems widely used for research on testing and debugging in SPL systems [35]. Our results show that by using CLUE, developers could find the bugs earlier with up to 88% less effort than the existing approaches. Additionally, for the same budgets, CLUE can help to detect many more bugs than the other approaches. Indeed, by using only 10% of test cases in the original test suite, CLUE can detect about 50% of the bugs in a buggy SPL system on average. Furthermore, CLUE not only improves the testing productivity by reducing the redundant tests that need to be executed but also maintains the debugging effort by preserving the fault localization performance, which ultimately improves the quality assurance process's productivity for SPLs.

In brief, this paper makes the following contributions:

1. CLUE: A novel test reduction approach for improving test productivity in SPL systems.
2. An extensive experimental evaluation showing the performance of CLUE over the state-of-the-art methods for SPL test reduction.

The rest of this paper is organized as follows. Section 2 reviews the related studies. Section 3

provides the basic concepts and our method to represent tests for reduction. Section 4 describes our SPL test reduction approach. Section 5 introduces the evaluation methodology for SPL test reduction approaches while Section 6 shows the experimental results. Finally, Section 7 concludes the paper.

## 2. RELATED WORKS

**Test suite reduction for non-configurable systems.** Multiple approaches have been proposed for improving the productivity of software testing and saving testing efforts [3, 12, 14, 17, 18, 26, 51, 52]. They can be divided into three main directions: test minimization, selection, and prioritization. In particular, test minimization [17, 26] reduces the cost of testing by permanently eliminating redundant test cases. Test selection [12, 25, 45, 47] focuses on reducing the size of the original test suite by selecting a subset of test cases to test a modified version. Meanwhile, test prioritization [3, 21, 49] aims to order the test cases for earlier satisfying some specific criteria.

For instance, Horgan et al. [26] applied linear programming to the test case minimization problem in implementing a data-flow-based testing tool, ATAC. In another research, Chen et al. [17] defined essential test cases as the opposite of redundant test cases and then applied the GE and GRE heuristics to minimize their test suites.

For selecting tests, that should be rerun after a modification, test selection approaches [25, 45, 47] often collect and analyze dependencies statically or dynamically. Dependencies can be collected at various granularity levels, and previous research has mostly focused on dynamic techniques. Rothermel and Harrold [47] proposed one of the first dynamic test selection techniques for C programs based on basic block-level analysis. Harrold et al. [25] later proposed to handle object-oriented features and adapt basic-block-level test selection for Java programs. Static test selection techniques [45] have been proposed in the past, but their effectiveness and efficiency were largely unexplored.

The test prioritization problem was initially defined by the Elbaum and Rothermel team [46]. In its early stages, they focused on white-box testing of test prioritization [21, 46], which prioritizes test cases based on program coverage and utilizes a greedy algorithm for ordering the execution. Wang et al. [49] introduce a cluster-based adaptive test case prioritization approach, which can add the new adaptive adjustment content in pre-prioritization. With the development of a deep learning model, TCP-Net [3] is proposed for using source code-related features, test case metadata, test case coverage information, and test case failure history to learn a high dimensional correlation between source files and test cases to prioritize tests.

These approaches are specialized for non-configurable systems, unaware of the similarities and differences of products in a family. In addition, they do not consider the feature interactions, which are inherent characteristics of the SPL systems. Thus, these approaches cannot be directly applied to reduce test suites of an SPL system. Different from these studies, CLUE focuses on specific characteristics of the SPL systems to address the redundant testing problem in this context.

**Test suite reduction for configurable systems.** To improve the testing productivity of the configurable systems, several approaches to configuration selection [20, 24] and configuration prioritization [7, 8, 36] have been proposed. For example, Al-Hajjiaji et al. [7, 8] select the next configurations for testing based on the similarity of the configurations with the previously selected ones. Nguyen et al. [36] prioritize configurations based on their number of potential bugs, which are measured by analyzing the feature interactions of the system. Moreover, to reduce the low-quality tests that are coincidental correctness and negatively impact FL results, Nguyen et al. introduced solutions for detecting and removing them at both the product level [38] and test-case level [39].

**Quality assurance for configurable systems.** Configurable systems create a mechanism to flexibly tailor products to customers' needs. Unfortunately, the large number of features, as well as their mutual interactions, make their quality notoriously challenging to assure. To guarantee the quality of configurable systems, there are various studies about variability-aware analysis for type checking [28, 29], testing [48, 50], and control/data flow analysis [15, 29]. Moreover, exhaustively testing SPL systems to detect faults is also extremely challenging. Therefore, various approaches [16, 20, 24] were also proposed to effectively test these systems.

## 3. TEST REPRESENTATION

A software product line (SPL) system is a family of *products* that share a common code base, and each product is distinguished from the others in terms of its selected *features* [9]. To test an SPL system $\mathfrak{S}$, a set of products $P = \{p_1, p_2, ..., p_n\}$ is often systematically *sampled/selected*; each product $p_i \in P$ has its own test suite $T_i$. Overall, to test $\mathfrak{S}$, we need to execute a huge set of tests $\mathcal{T} = \{t_{11}, ..., t_{nk}\}$, in which $t_{ij} \in T_i$ is a test case of product $p_i$.

In practice, multiple tests in $\mathcal{T}$ could be similar to some extent. For example, two tests $t_{i1}$ and $t_{j1}$ of two products $p_i$ and $p_j$ could be similar if they all target to validate the shared functionalities in these products. Executing these similar tests repeatedly does not provide more information about the correctness of the system but could consume a large amount of effort and resources. Thus, minimizing the test suite by selecting representative tests to reduce the efforts of redundantly executing similar test cases while retaining the fault detection rate is necessary.

In the existing studies for non-configurable code [21, 46, 49], the similarities of the tests are often measured based on their *execution profiles*, which specify whether a program element is executed (or not) when the tests run. For example, a program has three statements $s_1$, $s_2$, and $s_3$. If the execution profile of this program with test $t$ is $v = \langle a_1 = 1, a_2 = 0, a_3 = 1\rangle$, this means that statements $s_1$ and $s_3$ are executed and statement $s_2$ is not executed when test $t$ runs. These approaches hypothesize that similar tests often execute similar sets of statements and, thus, have similar execution profiles. However, these approaches require that each test must be executed at least once to record its execution profile. Thus, testing resources are not managed efficiently at first test execution because of (possible) similar tests in the test suites. In addition, if there is a modification leading to a change in the control flow of the program and affecting the execution profiles of the tests, using previous execution profiles to select unique tests could be incorrect.

Besides, the *test scripts*, which are sequences of code statements/instructions, could also be leveraged to identify similar test cases. The tests whose similar scripts are similar. However, in the context of SPL systems, not only the scripts of the tests but also the invoked methods need to be considered. This is because different products of an SPL system are constructed from different sets of features. Thus, they could have some similar and some different methods. The same test $t$ could be applied to validate different products of the system, but this test could verify similar or different behaviors in these products.

For example, Figure 1a shows the script of a test case used for validating the method `undoUpdate` of the class `Account` in the system `BankAccount`. Method `undoUpdate` consists of the source code of all the products of this system since it is implemented by the `Base` feature (Figure 1b), which is compulsorily enabled in all the products. For testing the functionalities of `undoUpdate` in each product of the system, the test $t$ shown in Figure 1a is employed to test all sampled products of this system. `BankAccount` system has 8 features with one compulsory feature and the others being optional. By using the 4-wise sampling technique, there are a total of 34 sampled products.

```java
public void test() throws Throwable {
  Account account0 = new Account();
  boolean boolean0 = account0.undoUpdate((-1));
  assertTrue(boolean0);
}
```

(a) A test case for method `undoUpdate` of class `Account`

```java
boolean undoUpdate(int x) {
    int newBalance = balance - x;
    if (newBalance < OVERDRAFT_LIMIT)
        return false;
    balance = newBalance;
    return true;
}
```

(b) Method `undoUpdate` implemented in the feature `Base`

```java
private boolean undoUpdate_wrappee_Base(int x) {
    int newBalance = balance - x;
    if (newBalance < OVERDRAFT_LIMIT)
        return false;
    balance = newBalance;
    return true;
}

boolean undoUpdate(int x) {
    int newWithdraw = withdraw;
    if (x < 0) {
        newWithdraw -= x;
        if (newWithdraw < DAILY_LIMIT)
            return false;
    }
    if (!undoUpdate_wrappee_Base(x))
        return false;
    withdraw = newWithdraw;
    return true;
}
```

(c) Method `undoUpdate` implemented in feature `DailyLimit`

Figure 1: A test case and two variants of method `undoUpdate` in the `BankAccount` system.

Intuitively, it is unnecessary to repeatedly execute this test 34 times in all 34 sampled products of the system. However, if this test is executed by only one product and skipped in the others, it could fail to validate the other variant of the method `undoUpdate` shown in Figure 1c. Specifically, the feature `DailyLimit`, which is an optional feature of `BankAccount`, implements another variant of the method `undoUpdate` to add some features. For the products disabling `DailyLimit`, the method `undoUpdate` of `Base` in Figure 1b is included in the products' source code. Meanwhile, for the products enabling `DailyLimit`, its method `undoUpdate` in Figure 1c is included in the products' source code. Therefore, *to measure the similarity of the test cases of an SPL system, both test scripts and the related methods are important.*

To precisely measure the similarity between test cases, we represent them by their scripts and related methods. For more details, the test script specifies the testing scenarios, inputs, and expected outputs; meanwhile, the behaviors the test aims to validate are implemented in the related methods. For a test $t$, its test script and related methods are represented in a *test entry.*

**Definition 1.** (Test entry) For a test $t \in T_i$ of the product $p_i$, the corresponding test entry is a pair $e = \langle t_s, M \rangle$, where $t_s$ is the test script of $t$ and $M = \{m_1, ..., m_k\}$ are the implementations of the test related methods in the product $p_i$. Particularly, a method $m_j \in M$ could be invoked by the test $t$ or by another method $m_k \in M$.

**Definition 2.** (Test reduction for SPL system) Given an SPL system $\mathfrak{S}$ with the set of test entries $\mathcal{E} = \{e_{11}, ..., e_{nk}\}$, where $e_{ij}$ is a test entry of product $p_i$, the test reduction problem is to output a set of entries $\mathcal{E}' \subset \mathcal{E}$ which retains a specified criteria $\mathcal{C}$, such as coverage, fault detection rate, etc. Specifically, $\varphi(\mathfrak{S}, \mathcal{E}', \mathcal{C}) = \varphi(\mathfrak{S}, \mathcal{E}, \mathcal{C})$, where $\varphi$ is a function measuring the specific criteria $\mathcal{C}$ in the system $\mathfrak{S}$ with the set of test entries $\mathcal{E}'$ (or $\mathcal{E}'$).

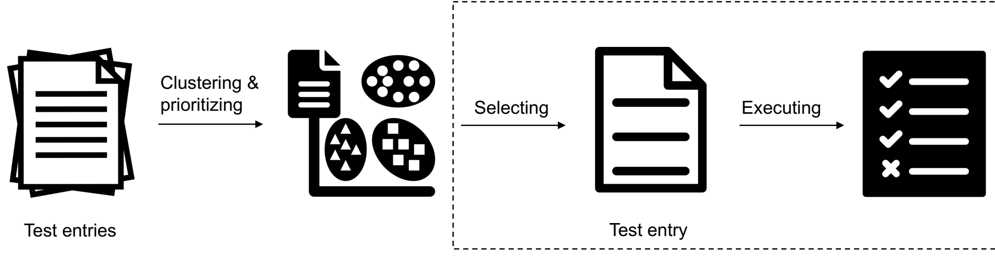



Figure 2: Approach overview.

## 4. CLUE: AN SPL TEST REDUCTION APPROACH

To efficiently and effectively test an SPL system, CLUE aims to select test entries that are distinctive and cover diverse behaviors of the system. In other words, CLUE focuses on selecting a subset of test cases that are different from each other in terms of both their test scripts and the related methods' implementations. Meanwhile, the test entries that are *similar* to the selected one are eliminated to reduce the waste of testing efforts.

Figure 2 shows the approach overview of CLUE. Particularly, for a set of test entries $\mathcal{E}$ of an SPL system $\mathfrak{S}$, CLUE *clusters* test entries into distinctive groups containing similar tests. In each cluster, the test entries are prioritized according to the number of feature interactions [9] in the related products. The reason is that most of the bugs in SPL systems are caused by undesirable interactions of the system features [1, 7, 37]. Thus, executing the tests in the product covered more interactions could increase the likelihood of revealing the bugs earlier. After that, CLUE continuously selects and executes the top-prioritized test in each cluster that has not been executed until a bug is revealed (a test is failed) or the effort budget is reached. The detailed algorithm of test selection of CLUE is demonstrated in Algorithm 1.

### 4.1. Test clustering

To determine distinctive groups of similar tests in the given set of test entries $\mathcal{E}$, we employ a clustering algorithm to group similar tests together into a distinct group (line 3, Algorithm 1). In fact, the number of groups the test entries $\mathcal{E}$ should be divided into is unknown in advance. Thus, the bottom-up clustering algorithms [5, 23], which start with individual data points and progressively group them into larger clusters if they are "close" enough, could be more suitable for our needs.

In this work, to capture the meaning of a test entry $e \in \mathcal{E}$, CLUE employs Word2Vec [34], which is a widely-used embedding technique in SE [19], to encode the content of $e$. Next, the *Hierarchical Agglomerative Clustering* algorithm [5], which is one of the most popular bottom-up clustering algorithms, is adopted to cluster test entries in $\mathcal{E}$. At the start, this algorithm considers each test entry as an individual cluster. In each iteration, the algorithm identifies and merges the two closet clusters into a single cluster. The distances of the clusters are measured by the distance metric, such as Euclidean distance, and they are updated after each iteration.

**Algorithm 1:** Test selection algorithm of CLUE

```
1  Function SelectTests(𝒯, effortBudget = −1)
2      selected_tests ← ∅
3      𝒞 ← clustering(𝒯)
4      for C ∈ 𝒞 do
5          PrioritizeTests(C)
6      end
7      searchStop ← False
8      while ¬searchStop do
9          for C ∈ 𝒞 do
10             t ← SelectAnUntestedTest(C)
11             r ← ExecuteTest(t)
12             selected_tests.add(t)
13             if r == "failed" and effortBudget == −1 then
14                 searchStop ← True
15                 break
16             end
17             if effortBudget ≠ −1 and cost(selected_tests) ≥ effortBudget then
18                 searchStop ← True
19                 break
20             end
21         end
22     end
23     selected_tests ← RemoveCoincidentalCorrectness(selected_tests)
24     return selected_tests
```

### 4.2. Test prioritizing

After clustering tests, each cluster contains similar test entries, which are tests of one or several products of the system. To increase the possibility of detecting bugs, we prioritize tests in each cluster by the number of feature interactions [22, 36, 37] in the corresponding products (lines $4-6$, Algorithm 1). The reason is that most of the bugs in SPL systems are caused by the interactions of the features [1, 37]. Thus, the products containing more feature interactions should be prioritized to test earlier.

Without loss of generality, let $C$ be a cluster containing two similar test entries $e_{ij}$ and $e_{mk}$. Particularly, the test $t_j$ of the product $p_i$ is similar to the test $t_k$ of the product $p_m$, and the implementation of the related methods in these two products is also similar. Testing product $p_i$ against $t_j$ is prioritized over testing $p_m$ against $t_k$, i.e., $e_{ij}$ has higher priority in the cluster $C$ than $e_{mk}$, if $p_i$ contains more feature interactions than $p_m$.

In a product of an SPL system, an interaction between two features exists if the presence of one feature affects the behaviors of the other. Features could interact with the others in multiple ways, such as a feature could influence/modify the other's behaviors, or the output of one feature is the input for the others, etc. Different kinds of feature interaction have been discussed in the literature [9, 10, 36, 37]. For instance, Nguyen et al. [36] identify the interaction of two features via their shared entities. If the two features share an entity, such as a global variable, then the appearance of a feature could change the value of the shared entity and then affect the manner of the other feature. In another research, VarCop [37]

detects feature interaction by analyzing how features impact the other via control/data dependencies.

In general, the interactions of the features in a product could be identified by analyzing the control/data dependencies of the program entities among the features [36, 37]. However, it is prohibitively expensive to conduct inter-procedure control/data dependence analysis. In this work, we aim to build an efficient test selection approach. Thus, the feature interactions in a product are estimated by the combinations of features in the products. The more feature combinations the product covers, the more feature interactions could be in that product [7, 32].

### 4.3. Test selecting

After clustering and prioritizing test entries in each cluster, CLUE iteratively selects a test in each cluster and executes the test in the corresponding product of the system (lines $7-22$, Algorithm 1). Depending on the setting of the users, CLUE could stop searching if a bug in the system is detected (i.e., the first failed test) or the effort budget is reached. The effort budget could be the test suite size, testing time, etc. Note that, in Algorithm 1, `effortBudget` is $-1$, which means the budget is not set, and the algorithm is in the early stop setting, i.e., stops searching right after a selected test fails. This is also the default setting of the algorithm.

Furthermore, to support the debugging process after a bug is detected, we remove the *coincidental correct* tests (line 23, Algorithm 1), which could negatively affect fault localization (FL) performance. Specifically, coincidental correct tests, are tests executing the fault but cannot reveal the failures [31, 38, 39]. These tests provide misleading indications about the execution of faults and negatively affect the performance of the FL technique, such as spectrum-based fault-localization (SBFL) techniques [43], measures the suspiciousness scores of the statements by the number of failed/passed tests executed by the statements.

A passed test is considered coincidental correctness if its execution profile is similar to a failed test's execution profile. Specifically, let $t_p$ and $t_f$ be passed and failed tests, respectively. Let $E_p$ and $E_f$ be the execution profiles of the two tests. The passed test $t_p$ is considered to be coincidentally passed if it executes a similar set of statements executed by $t_f$, i.e., $E_p$ is similar to $E_f$. The reason is that the fault has been executed by the failed test $t_f$ or the faulty statement(s) is in $E_f$. If $E_p$ is similar to $E_f$, $E_p$ could contain faulty statements. In other words, $t_p$ could also execute the faults as $t_f$ does. In this work, we do not use the test information of $t_p$ for the FL process if $sim(E_p, E_f) \geq \theta$. CLUE employs Jaccard [40] to measure the similarity of two sets and empirically sets the threshold $\theta = 0.8$.

## 5. EXPERIMENTAL METHODOLOGY

### 5.1. Research questions

For evaluation, we seek to answer the following research questions.

- **RQ1: Performance Comparison.** *How effective is our test reduction approach compared with the baseline approaches?*
- **RQ2: Fault localization analysis.** *How does our approach affect the fault localization (FL) results of the different SBFL metrics?*

Table 1: Dataset overview [35].

| System | #Features | #Products | #Tests | #Buggy versions |
|---|---|---|---|---|
| GPL | 27 | 99 | 8,514 | 372 |
| ZipMe | 13 | 25 | 6,262 | 304 |
| Email | 9 | 27 | 2,215 | 126 |
| BankAccount | 8 | 34 | 621 | 383 |
| ExamDB | 8 | 8 | 1,041 | 263 |
| Elevator | 6 | 18 | 2,553 | 122 |

- **RQ3: Clustering algorithm analysis.** *How do different clustering algorithms contribute to* CLUE*'s performance?*
- **RQ4: Parameter analysis.** *How do different parameters, including clustering distance thresholds and numbers of clusters, contribute to* CLUE*'s performance?*

### 5.2. Dataset

We conducted experiments on a public dataset of buggy SPL systems [35]. Currently, this is the only public dataset containing the versions of SPL systems that are affected by variability bugs and have been found through testing. To construct a large benchmark of variability bugs, Ngo et al. [35] proposed the bug generation process includes three main steps: Product Sampling and Test Generating, Bug Seeding, and Variability Bug Verifying. First, for an SPL system, a set of products is systematically sampled by sampling techniques such as t-wise or one-enabled/one-disabled algorithm [32]. To inject a fault into the system, a random modification is applied to the system's original source code by using a mutation operator, e.g., Arithmetic Operator Replacement or Logical Operator Replacement. In the last step, each generated bug is verified to ensure that the fault is a variability bug and caught by the tests.

Table 1 shows the detailed information of the dataset that we used in the experiments of this research. In total, there are six systems containing from 6 to 27 features. Among these systems, there are 1,570 buggy versions, of which 338 contain one bug each, and 1,232 contain multiple bugs each. For each buggy version of the SPL system, the system is sampled by 4-wise combinatorial testing, and each sampled product is tested against a test suite. The total number of tests used to validate each buggy version ranged from 621 tests in the BankAccount system to 8,514 tests in the GPL system.

### 5.3. Experimental procedure and evaluation metrics

#### 5.3.1. Experimental procedure

*Baselines:* We compared the performance of CLUE with the sate-of-the-art SPL test reduction approach, *Similarity-based* prioritization [7]. Also, to evaluate the complexities of the problem, we compare our approach with *Random-based* prioritization.

- *Random-based* prioritization: Test entries are selected in a random order.
- *Similarity-based* prioritization [7]: All the tests of the product with the maximum number of features are selected to execute first. Next, the tests of the product with the lowest feature similarity with the previously chosen products are validated.

*Procedures:* We evaluate the performance of the approaches in two settings:

- *Early-stop* setting: All the approaches gradually select and execute the tests of the system until a bug is detected. In other words, when a test is failed, all the approaches stop searching for the next tests.
- *Effort-budget* setting: All the approaches select and execute the same number of tests according to the given effort budget, i.e., test suite size. In this work, we evaluate the performance of the approaches when they select the different $k$ percentage of tests compared to the original test suites, $k = \{5\%, 10\%, 20\%, 50\%, 80\%\}$.

### 5.3.2. Evaluation metrics

To evaluate the SPL test reduction approaches, we measure the number of tests (*#Tests*) and *Fault Detected Rate (FDR)*. Particularly, *#Tests* specifies how many tests are selected by a test reduction approach to reveal the bug in the system. The smaller the number of tests, the better the approach. *Fault Detected Rate (FDR)* measures the percentage of bugs detected by the selected test suite. Given an SPL system containing $n$ bugs, a test reduction approach $A$ selects a test suite $T \in \mathcal{T}$, and there are $k$ bugs detected by the failed tests in $T$. The FDR of this approach is $FDR = \frac{k}{n}$. The higher the FDR, the more effective the approach.

To evaluate how practical the SPL test reduction approaches are in supporting the debugging process, we compare the fault localization (FL) performance of spectrum-based FL (SBFL) techniques using the test information of the test suite selected by each approach. Specifically, SBFL techniques measure the suspiciousness score of each statement in the program and then return a list of statements ordered by their suspiciousness scores. The statement which is the most suspicious will be at the top of the list. Following the existing FL studies for SPL [11, 35, 37], the FL performance is measured in *Rank*. Rank indicates the position of the buggy statements in the resulting lists of the FL techniques. The lower the Rank of buggy statements, the more effective the approach. If multiple statements have the same score, buggy statements are ranked last. Moreover, for the cases of multiple bugs, we measured *Rank*s of the first buggy statement (*best Rank*) in the lists.

## 6. EXPERIMENTAL RESULTS

### 6.1. Performance comparison

### 6.1.1. Early-stop setting

Table 2 shows the performance of the SPL test reduction approaches in the early-stop setting. As seen, CLUE needs to execute the smallest number of tests to obtain a similar FDR compared to the other baselines. For example, in each buggy version of `BankAccount`, CLUE only needs to execute 117 tests to find the failed test. In comparison, these figures for Random-based and Similarity-based approaches are 142 and 315 tests, respectively. Notably, for each buggy version of `ZipMe`, CLUE can detect a bug by executing only 4% of tests of the original test suite, which is much less than the Random-based approach with 84% and the Similarity-based approach with 88%. *These results demonstrate that by using* CLUE*, developers could earlier find bugs with much less effort than the other approaches.*

Table 2: Test reduction performance comparison in the *early-stop* setting.

| | **Original** | | **Random-based** | | **Similarity-based** | | **Clue** | |
|---|---|---|---|---|---|---|---|---|
| System | #Tests | FDR | #Tests | FDR | #Tests | FDR | #Tests | FDR |
| BankAccount | 621 | 0.85 | 142 | 0.59 | 315 | 0.59 | 117 | 0.58 |
| Email | 2,215 | 0.75 | 450 | 0.60 | 1,039 | 0.60 | 81 | 0.59 |
| Elevator | 2,553 | 0.70 | 266 | 0.57 | 298 | 0.52 | 100 | 0.56 |
| ExamDB | 1,041 | 0.54 | 527 | 0.54 | 415 | 0.54 | 175 | 0.54 |
| GPL | 8,514 | 0.81 | 1,360 | 0.61 | 4,265 | 0.61 | 453 | 0.62 |
| ZipMe | 6,262 | 0.63 | 1,450 | 0.49 | 1,953 | 0.49 | 226 | 0.49 |

Table 3: FDR of the SPL test reduction approaches with different *effort budgets*.

| System | | Test Size | | | | | |
|---|---|---|---|---|---|---|---|
| | | 100% | 80% | 50% | 20% | 10% | 5% |
| BankAccount | Random-based | 0.85 | 0.78 | 0.58 | 0.30 | 0.14 | 0.10 |
| | Similarity-based | 0.85 | 0.84 | 0.41 | 0.09 | 0.06 | 0.04 |
| | Clue | 0.85 | 0.79 | 0.66 | 0.40 | 0.23 | 0.20 |
| Email | Random-based | 0.75 | 0.72 | 0.61 | 0.31 | 0.17 | 0.07 |
| | Similarity-based | 0.75 | 0.63 | 0.50 | 0.05 | 0.05 | 0.05 |
| | Clue | 0.75 | 0.75 | 0.75 | 0.69 | 0.67 | 0.64 |
| Elevator | Random-based | 0.70 | 0.69 | 0.67 | 0.53 | 0.36 | 0.21 |
| | Similarity-based | 0.70 | 0.59 | 0.59 | 0.42 | 0.37 | 0.37 |
| | Clue | 0.70 | 0.70 | 0.67 | 0.64 | 0.59 | 0.43 |
| ExamDB | Random-based | 0.54 | 0.44 | 0.28 | 0.11 | 0.04 | 0.03 |
| | Similarity-based | 0.54 | 0.54 | 0.29 | 0.17 | 0.17 | 0.17 |
| | Clue | 0.54 | 0.54 | 0.52 | 0.34 | 0.19 | 0.13 |
| GPL | Random-based | 0.81 | 0.79 | 0.70 | 0.45 | 0.25 | 0.12 |
| | Similarity-based | 0.81 | 0.66 | 0.44 | 0.11 | 0.06 | 0.02 |
| | Clue | 0.81 | 0.78 | 0.70 | 0.72 | 0.54 | 0.50 |
| ZipMe | Random-based | 0.63 | 0.58 | 0.47 | 0.26 | 0.11 | 0.07 |
| | Similarity-based | 0.63 | 0.54 | 0.35 | 0.33 | 0.01 | 0.01 |
| | Clue | 0.63 | 0.63 | 0.63 | 0.60 | 0.57 | 0.39 |

The Similarity-based prioritization approach aims to select products by maximizing the diversity of feature combinations. Then, it tests each selected product at a time. For a selected product $p_i$, this approach continuously executes each test in the product's test suite $T_i$. Consequently, all the similar tests in $T_i$ and the following selected products are still redundantly executed until the bug is found. Although this approach is aware of feature interactions for prioritizing testing a product, its performance could be even worse than the Random-based approach.

Meanwhile, Clue can find bugs earlier with fewer tests being attempted since it selects tests of the whole system regarding their diversity and distinction. Specifically, by appropriately representing tests and clustering them, Clue can group similar tests together. Similar

tests in a cluster could be the tests of one or several products of the system. This helps CLUE avoid repeatedly executing similar test cases as Similarity-based prioritization does. Moreover, CLUE considers the feature interactions in selecting tests in each cluster instead of randomly selecting them. This increases the possibility of encountering the bugs by CLUE. Therefore, CLUE could detect bugs far earlier by a much smaller number of tests compared to the other baselines.

Note that the FDR of all the approaches in this setting is similar since they all stop searching when a failed test is found. In other words, the selected test suites of each approach contain only one failed test for each buggy SPL system. As a result, their fault detection rates are quite similar.

#### 6.1.2. Effort-budget setting

Table 3 shows the FDR of the approaches when selecting test suites with different effort budgets. Overall, for the same budgets, CLUE can help detect more bugs than the other approaches. On average, by using only 10% of the test cases in the original test suite, CLUE can detect about 50% of the bugs in a buggy SPL system. Meanwhile, with the same number of tests, the Random-based prioritization approach can detect about 20% of the bugs, and the Similarity-based prioritization approach can detect only 12% of the bugs. To detect a similar number of bugs, Random-based and Similarity-based approaches need to execute about 50% of the tests in the original test suites.

Moreover, by using CLUE, developers could save 20% of testing efforts while preserving nearly the same FDR of the original test suites of the SPL systems. This means that, by using the test suites selected by CLUE, only 80% of test cases need to be executed to detect all bugs that are covered by the original test suites in each buggy system. Especially, for 4 out of 6 systems, including `Email`, `Elevator`, `ExamDB`, and `ZipMe`, CLUE can save up to 50% of testing efforts. Indeed, CLUE attempts to select divergent test entries in each selection iteration. This increases coverage of selected tests of CLUE and therefore increases its FDR.

The FL performance of SBFL using the test information after executing the test suites selected by the SPL test reduction approaches is shown in Table 4. As seen, SBFL obtains the best performance with the tests selected by CLUE, compared to the baselines. For instance, by selecting only 50% of test cases in `Email` systems, CLUE can help SBFL to localize the buggy statements at Rank $9^{th}$, which is equivalent to FL results using the whole test suite. Meanwhile, for selecting the same 50% of the test cases, the SBFL results using the selected tests of Random-based and Similarity-based prioritization approaches are $26^{th}$ and $63^{rd}$, respectively. These figures illustrate that these two approaches could save test efforts by reducing tests, but after that, developers need to spend much more effort on debugging. Meanwhile, CLUE *not only helps to save testing efforts by reducing redundant tests but also maintains the debugging effort by preserving the FL results.*

### 6.2. Fault localization analysis

Table 5 shows the performance of the 5 most popular SBFL metrics [42] using the test suites selected by the SPL test reduction approaches with an effort budget of 80% of the original test suites. In general, for all 5 SBFL metrics, the FL performance using the selected tests of CLUE is quite stable compared to those results using the original test suites.

Table 4: FL performance (by *Rank*) of SBFL using the test suites selected by the SPL test reduction approaches.

| System | | Test Size | | | | | |
|---|---|---|---|---|---|---|---|
| | | 100% | 80% | 50% | 20% | 10% | 5% |
| BankAccount | Random-based | 5 | 6 | 16 | 41 | 57 | 58 |
| | Similarity-based | 5 | 5 | 32 | 67 | 70 | 58 |
| | Clue | 5 | 5 | 13 | 29 | 43 | 45 |
| Email | Random-based | 9 | 11 | 26 | 123 | 180 | 208 |
| | Similarity-based | 9 | 19 | 63 | 231 | 227 | 188 |
| | Clue | 9 | 9 | 9 | 19 | 24 | 31 |
| Elevator | Random-based | 14 | 15 | 16 | 73 | 176 | 271 |
| | Similarity-based | 14 | 40 | 39 | 79 | 108 | 120 |
| | Clue | 14 | 12 | 14 | 28 | 71 | 128 |
| ExamDB | Random-based | 3 | 44 | 110 | 172 | 185 | 180 |
| | Similarity-based | 3 | 3 | 98 | 111 | 121 | 120 |
| | Clue | 3 | 3 | 8 | 97 | 122 | 133 |
| GPL | Random-based | 7 | 8 | 49 | 304 | 555 | 709 |
| | Similarity-based | 7 | 113 | 408 | 824 | 881 | 859 |
| | Clue | 7 | 10 | 49 | 43 | 167 | 194 |
| ZipMe | Random-based | 122 | 130 | 267 | 639 | 891 | 958 |
| | Similarity-based | 122 | 224 | 458 | 469 | 1123 | 1092 |
| | Clue | 122 | 124 | 124 | 136 | 157 | 352 |

Specifically, for 3 out of 6 systems, including `BankAccount`, `Email`, and `ExamDB`, the FL performance in all 5 metrics using the test suites selected by Clue and the original test suites are equivalent.

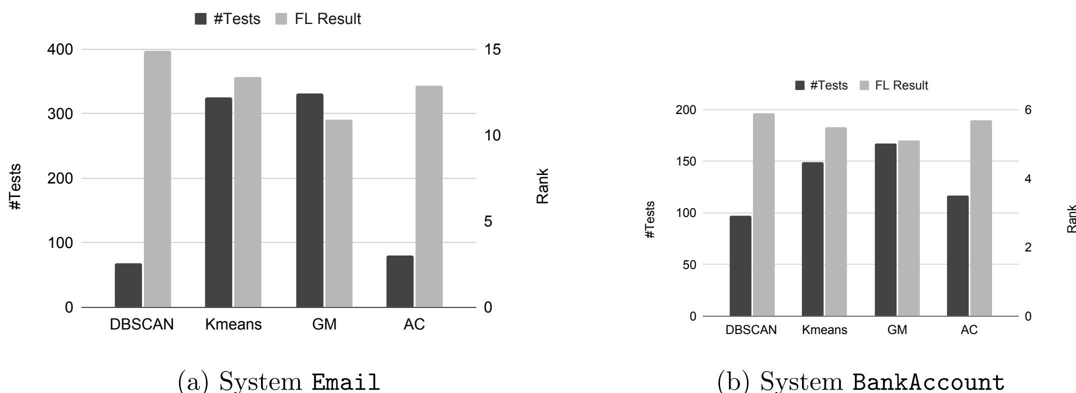
Interestingly, the FL results of the buggy versions of `Elevator are even improved when using the selected test suites of Clue`. For instance, using the whole original test suite, Tarantula and Barinel rank the buggy statements of this SPL system at Rank $14^{th}$. Meanwhile, by using the testing information of the selected test cases, Clue helps to improve these results by 14%, the Ranks returned by Tarantula and Barinel are $12^{th}$. This is reasonable because the FL techniques [4, 11, 37] often evaluate the suspiciousness of the code statements based on the number of passed and failed tests that the statements executed. By Clue, tests are iteratively selected in different clusters. This leads to the selected tests being distinctive. If the selected tests are distinctive, their execution profiles are diverse and unique. In other words, the suspiciousness scores of the statements are better distinguished since the numbers of tests each statement executed are different. Therefore, the FL performance [44] can be improved. Meanwhile, if the selected tests are similar, their execution profiles could be identical. It is difficult for FL techniques to find the bugs since the numbers of passed and failed tests executed by the code statements are similar. Moreover, Clue also detects and removes coincidental correct tests before conducting FL. As a result, the misleading indications of faults could be eliminated, and lead to the improvement of FL performance.

Table 5: FL performance (by *Rank*) of different SBFL metrics with the test suites selected by the approaches.

| System | | Tarantula | Op2 | Ochiai | Barinel | Dstar |
|---|---|---|---|---|---|---|
| BankAccount | Random-based | 6 | 5 | 6 | 6 | 6 |
| | Similarity-based | 5 | 4 | 4 | 5 | 4 |
| | CLUE | 5 | 4 | 4 | 5 | 4 |
| | Original test suite | 5 | 4 | 4 | 5 | 4 |
| Email | Random-based | 11 | 8 | 10 | 11 | 10 |
| | Similarity-based | 19 | 9 | 18 | 19 | 18 |
| | CLUE | 9 | 7 | 8 | 9 | 8 |
| | Original test suite | 9 | 7 | 8 | 9 | 8 |
| Elevator | Random-based | 15 | 11 | 9 | 15 | 9 |
| | Similarity-based | 40 | 23 | 40 | 40 | 40 |
| | CLUE | 12 | 10 | 8 | 12 | 8 |
| | Original test suite | 14 | 10 | 8 | 14 | 8 |
| ExamDB | Random-based | 44 | 20 | 44 | 44 | 44 |
| | Similarity-based | 3 | 3 | 3 | 3 | 3 |
| | CLUE | 3 | 3 | 3 | 3 | 3 |
| | Original test suite | 3 | 3 | 3 | 3 | 3 |
| GPL | Random-based | 8 | 6 | 6 | 8 | 5 |
| | Similarity-based | 113 | 37 | 112 | 113 | 112 |
| | CLUE | 10 | 8 | 8 | 10 | 8 |
| | Original test suite | 7 | 5 | 5 | 7 | 5 |
| ZipMe | Random-based | 130 | 108 | 129 | 130 | 129 |
| | Similarity-based | 224 | 122 | 223 | 224 | 223 |
| | CLUE | 124 | 108 | 122 | 124 | 122 |
| | Original test suite | 122 | 106 | 120 | 122 | 120 |

Meanwhile, the performance of different SBFL metrics is unstable with the selected test suites of the Random-based and Similarity-based approaches. Particularly, Op2 obtains better results than the other SBFL metrics. For `Email`, with the selected tests of Similarity-based approach, Op2 localizes the buggy statements at the Rank of $9^{th}$, while the other metrics rank them at $18^{th} - 19^{th}$. In addition, by using the selected tests of the Random-based approach in `ExamDB`, buggy statements are ranked $20^{th}$ by Op2, while ranked $44^{th}$ by the others. Meanwhile, the FL results with the original test suites and the selected tests of CLUE are stable among all the experimental SBFL metrics.

Moreover, the FL performance using the tests selected by Random-based and Similarity-based approaches is much worse than CLUE and the original test suites in all 5 SBFL metrics. For example, by using the original test suites or the tests selected by CLUE, Ochiai pinpoints the buggy statements of `Email` at the Rank of $8^{th}$. However, using the selected tests of Random-based and Similarity-based approaches, the results of Ochiai for these buggy statements are $10^{th}$ and $18^{th}$, respectively. This demonstrates the poor performance of these two approaches in supporting the debugging process after a fault is detected.




(a) System `Email` (b) System `BankAccount`

Figure 3: Impact of different clustering algorithms on CLUE's performance.

### 6.3. Clustering algorithm analysis

In this experiment, we evaluate how CLUE works with different clustering algorithms. There are four main types of clustering algorithms, including density-based, distribution-based, centroid-based, and hierarchical-based. For each type of clustering algorithm, we select a representative technique and alternatively employ it in CLUE for clustering tests in the buggy versions of `Email` and `BankAccount` systems. The experimental results are measured in the early-stop setting. Specifically, the clustering techniques are Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [13], Gaussian Mixture (GM) [23], $K$-means [30], and Agglomerative clustering (AC) [5], respectively.

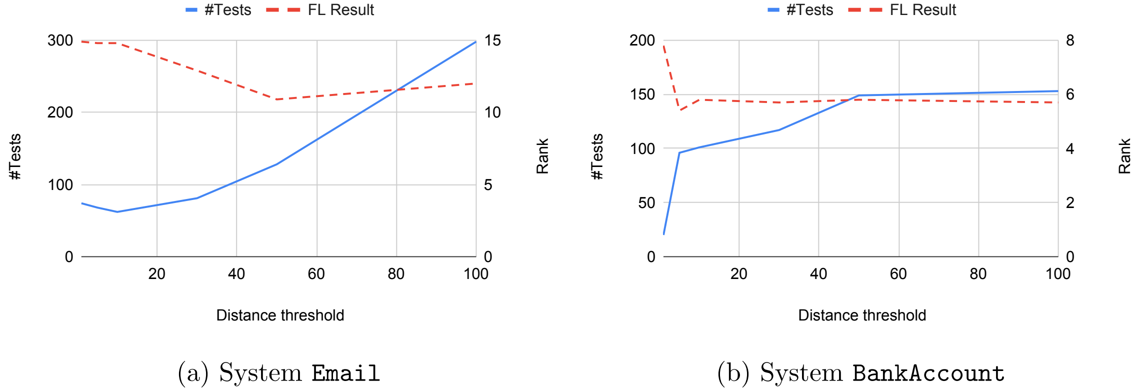
As seen in Figure 3, different clustering techniques have different affects on CLUE's performance. With DBSCAN, CLUE attempts the least number of tests, while with GM, CLUE attempts the highest number of tests to detect a bug in buggy systems. In contrast, with DBSCAN, the FL results are the worst, while the FL results are the best with the selected test suites of CLUE with GM. It is reasonable because CLUE with GM produces the largest test suites, which helps the FL technique have more information to localize the faults and obtain better performance.

In addition, the test suites selected by CLUE with AC are slightly larger than the results of CLUE with DBSCAN, yet much better than the results of CLUE with $K$-means and GM. For example, by AC algorithm, 81 tests are selected for the `Email` system, while these features of CLUE with $K$-means and GM are 325 and 332 tests, respectively. For the `BankAccount` system, with AC, 117 tests are selected, while with GM, 167 tests are selected by CLUE. Moreover, the FL result using the tests selected by CLUE with AC is up to 13% better than that of CLUE with DBSCAN. Therefore, *in order to balance between the test suite size and FL performance, AC is highly recommended for clustering test entries in* CLUE.

### 6.4. Parameter analysis

#### 6.4.1. Impact of different distance thresholds

Agglomerative clustering is a bottom-up approach that starts by considering each test entry as a single cluster and then progressively groups them into larger clusters if they




(a) System `Email` (b) System `BankAccount`

Figure 4: Impact of different distance thresholds on CLUE's performance.

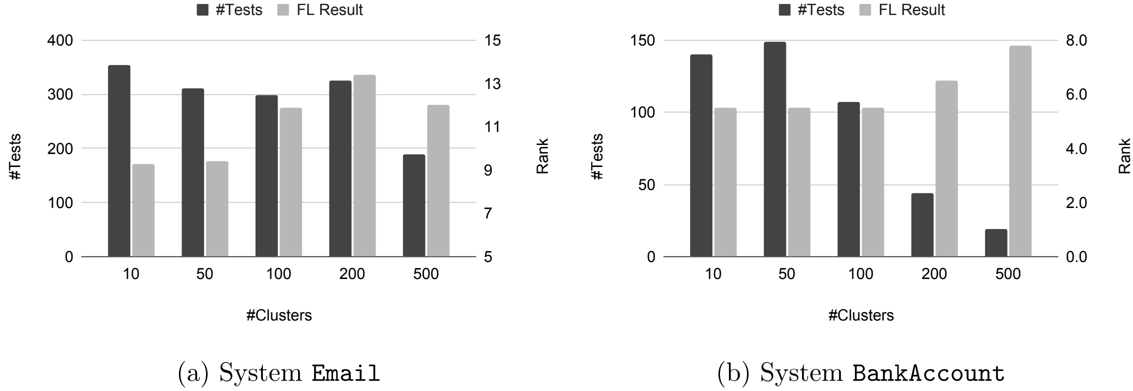
are closer than a distance threshold $d$. The distances among the clusters are measured by Euclidean metric. In this experiment, we gradually varied $d$ from 1 to 100 and applied clustering tests in the buggy versions of two systems, `Email`, and `BankAccount`.

Figure 4 shows that the larger the distance thresholds, the more tests need to be executed to find the first failed test. For example in the `Email` system, if the distance threshold $d = 5$, the selected test suite contains 68 tests. Meanwhile, if the distance threshold $d = 100$, the number of selected tests significantly increases to 298 tests. These figures in the `BankAccount` system are 96 tests when $d = 5$ and 153 tests when $d = 100$. This is because when the distance threshold is large, the similarity of tests is loosely measured to group them into a cluster. As a result, with a large distance threshold, the tests in each cluster are less similar, and the number of created clusters is smaller than when the distance threshold is small. In other words, with large distance thresholds, the clustering performance could be less precise in grouping similar tests together. This could lead CLUE to iterate over the clusters multiple times for selecting tests and result in a large selected test suite.

Moreover, with a large number of executed tests, the FL technique has more information to find the buggy statements and better rank them. For the `Email` system, the FL result when CLUE ith $d = 100$ is $12^{th}$, while this figure when CLUE with $d = 5$ is $15^{th}$. In our experiments, *the FL technique obtains the best performance when* $d = 50$. For instance, in the `Email` system, the number of selected tests is 128 tests, and the average Rank of the buggy statements with these tests is $11^{th}$. Meanwhile, the number of selected tests for the `BankAccount` system is 149 tests and the buggy statements are ranked at $6^{th}$.

### 6.4.2. Impact of different cluster numbers

$K$-means aims to partition test entries into pre-defined $k$ clusters. This experiment investigates how different numbers of clusters affect the performance of CLUE. As seen in Figure 5, the smaller the number of clusters, the larger the number of tests that need to be executed to find the first failed test. In addition, the larger the number of executed tests, the better FL performance. For example, if there are 10 clusters, $k = 10$, the number of selected tests is 354 tests for the `Email` system and 140 tests for the `BankAccount` system. In addition, with these selected tests, the Ranks of buggy statements in `Email` and `BankAccount`




(a) System `Email` (b) System `BankAccount`

Figure 5: Impact of different cluster numbers on CLUE's performance.

are about $9^{th}$ and $5^{th}$. Meanwhile, if there are 500 clusters, $k = 500$, these figures are 190 tests and bug Ranks at $12^{th}$ for `Email`, and 19 tests and bug Ranks at $8^{th}$ for `BankAccount`, respectively. This trending of results complies with the results about the impact of distance thresholds analyzed in Subsection 6.4.1.

### 6.5. Threats to validity

The main threats to the validity of our work consist of internal, external, and construct validity threats.

Threats to internal validity mainly lie in the correctness of the implementation of our approach. To reduce this threat, we carefully reviewed our code.

Threats to external validity are primarily associated with the benchmark used in our experiments. Although the dataset uses the systems widely used in the existing work, this dataset only contains artificial bugs of Java SPL systems, so we cannot extrapolate our findings to the real-world faults and SPL systems in different programming languages. To mitigate this threat, we chose the dataset containing a large number of buggy products with a diversity of artificial faults, and each product is tested by a large number of test cases. Also, the dataset contains both single-bug and multiple-bug buggy systems. Moreover, we also plan to collect and conduct experiments on more real-world variability bugs in larger SPL systems to evaluate our techniques.

Threats to construct validity mainly lie in the rationality of the assessment metrics. To reduce this threat, we chose the metrics that are widely used in the related studies [7, 36, 37].

## 7. CONCLUSIONS

In this paper, we introduce CLUE, a novel test reduction approach to improve the productivity of SPL testing. Based on the idea that similar tests often cover similar behaviors, CLUE clusters tests into distinctive groups, prioritizing them based on the prevalence of feature interactions, a common cause of defects within SPLs. The approach effectively ensures the execution of tests that are most likely to reveal defects. The experimental evaluation of CLUE on a dataset comprising six widely used SPL systems shows that CLUE outperformed

the state-of-the-art approaches for SPL test reduction. With CLUE, developers can detect defects earlier, requiring significantly less effort than existing approaches. Moreover, CLUE not only trims redundant tests but also maintains fault localization performance, contributing to enhanced quality assurance productivity for SPL systems.

## REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of The 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 421–432.

[2] I. Abal, J. Melo, Ş. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski, "Variability bugs in highly configurable systems: A qualitative analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 3, pp. 1–34, 2018.

[3] M. Abdelkarim and R. ElAdawi, "Tcp-net: Test case prioritization using end-to-end deep neural networks," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2022, pp. 122–129. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSTW55395.2022.00034

[4] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 88–99.

[5] M. R. Ackermann, J. Blömer, D. Kuntze, and C. Sohler, "Analysis of agglomerative clustering," *Algorithmica*, vol. 69, pp. 184–215, 2014.

[6] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake, "Incling: efficient product-line testing using incremental pairwise sampling," *ACM SIGPLAN Notices*, vol. 52, no. 3, pp. 144–155, 2016.

[7] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake, "Effective product-line testing using similarity-based product prioritization," *Software & Systems Modeling*, vol. 18, pp. 499–521, 2019.

[8] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake, "Similarity-based prioritization in software product-line testing," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*, 2014, pp. 197–206.

[9] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated, 2013.

[10] S. Apel, A. Von Rhein, T. ThüM, and C. KäStner, "Feature-interaction detection based on feature-based specifications," *Computer Networks*, vol. 57, no. 12, pp. 2399–2409, 2013.

[11] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, "Spectrum-based fault localization in software product lines," *Information and Software Technology*, vol. 100, pp. 18–31, 2018.

[12] M. Azizi and H. Do, "Retest: A cost effective test case selection technique for modern software development," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 144–154.

[13] H. Bäcklund, A. Hedblom, and N. Neijman, "A density-based spatial clustering of application with noise," *Data Mining TNM033*, vol. 33, pp. 11–30, 2011.

[14] C. Birchler, S. Khatiri, P. Derakhshanfar, S. Panichella, and A. Panichella, "Single and multi-objective test cases prioritization for self-driving cars in virtual environments," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–30, 2023.

[15] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "Spllift: Statically analyzing software product lines in minutes instead of years," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 355–364, 2013.

[16] I. Cabral, M. B. Cohen, and G. Rothermel, "Improving the testing and testability of software product lines," in *International Conference on Software Product Lines*. Springer, 2010, pp. 241–255.

[17] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, 1996.

[18] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, A. Corazza, and G. Antoniol, "Adequate vs. inadequate test suite reduction approaches," *Information and Software Technology*, vol. 119, p. 106224, 2020.

[19] Z. Ding, H. Li, W. Shang, and T.-H. P. Chen, "Can pre-trained code embeddings improve model performance? revisiting the use of code embeddings in software engineering tasks," *Empirical Software Engineering*, vol. 27, no. 3, pp. 1–38, 2022.

[20] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. De Almeida, "On strategies for testing software product lines: A systematic literature review," *Information and Software Technology*, vol. 56, no. 10, pp. 1183–1199, 2014.

[21] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, pp. 185–210, 2004.

[22] B. J. Garvin and M. B. Cohen, "Feature interaction faults revisited: An exploratory study," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 90–99.

[23] J. Goldberger and S. Roweis, "Hierarchical clustering of a mixture model," *Advances in Neural Information Processing Systems*, vol. 17, 2004.

[24] M. Greiler, A. van Deursen, and M.-A. Storey, "Test confessions: A study of testing practices for plug-in systems," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 244–254.

[25] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for java software," *ACM Sigplan Notices*, vol. 36, no. 11, pp. 312–326, 2001.

[26] J. R. Horgan and S. London, "A data flow coverage testing tool for c," in *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*. IEEE Computer Society, 1992, pp. 2–3.

[27] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*, 2012, pp. 46–55.

[28] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 3, pp. 1–39, 2012.

[29] J. Liebig, A. Von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 81–91.

[30] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern recognition*, vol. 36, no. 2, pp. 451–461, 2003.

[31] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, pp. 1–28, 2014.

[32] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 643–654.

[33] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: measuring interactions in highly-configurable systems," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 483–494.

[34] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013*, Y. Bengio and Y. LeCun, Eds., 2013.

[35] K.-T. Ngo, T.-T. Nguyen, S. Nguyen, and H. D. Vo, "Variability fault localization: a benchmark," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*, 2021, pp. 120–125.

[36] S. Nguyen, H. Nguyen, N. Tran, H. Tran, and T. Nguyen, "Feature-interaction aware configuration prioritization for configurable code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 489–501.

[37] T.-T. Nguyen, K.-T. Ngo, S. Nguyen, and H. D. Vo, "A variability fault localization approach for software product lines," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4100–4118, 2021.

[38] ——, "Detecting false-passing products and mitigating their impact on variability fault localization in software product lines," *Information and Software Technology*, vol. 153, p. 107080, 2023.

[39] T.-T. Nguyen and H. D. Vo, "Detecting coincidental correctness and mitigating its impacts on localizing variability faults," in *2022 14th International Conference on Knowledge and Systems Engineering (KSE)*. IEEE, 2022, pp. 1–6.

[40] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of The International Multiconference of Engineers and Computer Scientists*, vol. 1, no. 6, 2013, pp. 380–384.

[41] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *International Conference on Software Product Lines*. Springer, 2010, pp. 196–210.

[42] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating & improving fault localization techniques," *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03*, p. 27, 2016.

[43] ——, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 2017, pp. 609–620.

[44] A. Perez, R. Abreu, and A. Van Deursen, "A theoretical and empirical analysis of program spectra diagnosability," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 412–431, 2019.

[45] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby, "Chianti: A prototype change impact analysis tool for java," Rutgers University, Tech. Rep., 2003.

[46] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: an empirical study," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, 1999, pp. 179–188.

[47] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, pp. 173–210, 1997.

[48] L. R. Soares, J. Meinicke, S. Nadi, C. Kästner, and E. S. de Almeida, "Exploring feature interactions without specifications: A controlled experiment," *ACM SIGPLAN Notices*, vol. 53, no. 9, pp. 40–52, 2018.

[49] X. Wang and S. Zhang, "Cluster-based adaptive test case prioritization," *Information and Software Technology*, p. 107339, 2023.

[50] C.-P. Wong, J. Meinicke, L. Lazarek, and C. Kästner, "Faster variational execution with transparent bytecode transformation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[51] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[52] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

*Received on December 19, 2023*
*Accepted on April 17, 2024*