Empir Software Eng (2018) 23:1–51
DOI 10.1007/s10664-016-9494-9

CrossMark

Genetic Algorithm-based Test Generation for Software Product Line with the Integration of Fault Localization Techniques

Xuelin Li[1] · W. Eric Wong[1] · Ruizhi Gao[1] · Linghuan Hu[1] · Shigeru Hosono[2]

Published online: 5 February 2017
© Springer Science+Business Media New York 2017

Abstract In response to the highly competitive market and the pressure to cost-effectively release good-quality software, companies have adopted the concept of software product line to reduce development cost. However, testing and debugging of each product, even from the same family, is still done independently. This can be very expensive. To solve this problem, we need to explore how test cases generated for one product can be used for another product. We propose a genetic algorithm-based framework which integrates software fault localization techniques and focuses on reusing test specifications and input values whenever feasible. Case studies using four software product lines and eight fault localization techniques were conducted to demonstrate the effectiveness of our framework. Discussions on factors that may affect the effectiveness of the proposed framework is also presented. Our results indicate that test cases generated in such a way can be easily reused (with appropriate conversion) between different products of the same family and help reduce the overall testing and debugging cost.

Keywords Software product line · Genetic algorithm · Test generation · Debugging/fault localization · Coverage · EXAM score

1 Introduction

Software has become fundamental to our society and our everyday lives. Regardless of age, gender, occupation, or nationality, each one of us depends on software, either directly or

---

Communicated by: Franz Wotawa, Rui Abreu, T.H Tse and Birgit Hofer

✉ W. Eric Wong
ewong@utdallas.edu

[1] Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA

[2] Service Business Development Division, NEC Corporation, Tokyo, Japan

indirectly. Yet the disappointing truth is that software is far from defect-free and that very large sums of money are spent each year only to fix and maintain defective software. According to a National Institute of Standards and Technology report (NIST Report 2002), software bugs cost the U.S. economy an estimated $59.5 billion annually. This situation has deteriorated because not only is software larger and much more complicated than ever before, but the industry also suffers from a tighter schedule and reduced cost. On the other hand, the same study also found that more than one-third of these costs could be eliminated by an improved quality assurance infrastructure.

The concept of software product line (SPL) (Clements and Northrop 2001; Pohl et al. 2005; Bergey et al. 2010) has been adopted by companies such as NEC in order to improve the overall quality of their products and reduce time and efforts needed to produce high quality software. A selective suite of methods, tools, and techniques have been utilized to produce a collection of similar software products from a shared set of assets such as source code, programmers' knowledge and experience, and documents. More precisely, a group of software-intensive systems that share a common, managed set of features satisfying some specific needs of a particular market segment can be developed from a common set of core assets in a prescribed way (Clements and Northrop 2001). There are many advantages to using this approach. One is that companies can significantly save their capital investments. The other is that additional training and potential work time wasted on learning curve for new techniques and tools can also be dramatically reduced.

However, saving the initial implementation cost by reusing assets only reduces the overall cost to a certain extent. It is estimated that at least 60% of the cost of software development is spent on testing and debugging (Dustin et al. 2009; Vessy 1985). Unfortunately, at this moment, most of the techniques for testing and debugging software product lines are still performed individually with respect to each product (McGregor 2001; Perrouin et al. 2012; Perrouin et al. 2010; Pohl et al. 2005). In order to ensure the quality of products within a product line, all possible uses of each generic component, and preferably all possible product variants, need to be examined. This is neither efficient nor effective, as it requires the generation of a large number of test cases of which some may be "redundant" from the testing and debugging point of view. Additionally, the fact that the number of possible product variants grows exponentially with the number of variation points makes such approach very expensive (Engström and Runeson 2011).

Moreover, it is well-known that manual debugging is not only ineffective and inefficient but also fault-prone (Goel 1985; Liu et al. 2006; Xie et al. 2013a). As a result, those approaches that help automate the process of debugging have attracted the attention of both industry and academia. Software fault localization techniques, which provide a ranking of suspicious components (i.e. statements), have proven to be a promising branch of debugging activities that guarantee the quality of software by making it more reliable and maintainable. However, the techniques themselves have been recognized as some of the most expensive in debugging. When our goal is to debug an SPL containing a large number of variants, the process is more complicated and therefore even more expensive. Integrating various fault localization techniques into software product line engineering (SPLE) would improve the quality of the whole product family, however such integration has not yet been proposed in the literature.

To solve this problem, we use a genetic algorithm (GA)-based test generation technique which integrates fault localization metrics with a focus on how test cases generated for one product can be applied to test another product of the same family. Two different strategies using Euclidean distance and K-Means clustering, respectively, are proposed in our technique to help

generate new test cases. In our case study, we used four SPLs to demonstrate the effectiveness of our proposed technique. It has been demonstrated that test cases generated by our proposed technique achieve higher code coverage (statement, condition, and all-use coverage) and are more effective in locating bugs compared to those generated by random testing.

Background knowledge is introduced in Section 2. Section 3 provides the methodology of our proposed GA-based test generation technique. Section 4 presents two running examples showing how the proposed technique works. The subject programs, data collection as well as evaluation metrics are introduced in Section 5. Section 6 report our case studies along with collected data and corresponding analyses. Performance date regarding the proposed framework is presented in Section 7. Discussions on factors that may have an impact on the effectiveness of the proposed technique is introduced in Section 8. Related studies and threats to validity are presented in Section 9 and Section 10. Our conclusions and direction of future work can be found in Section 11.

2 Background

2.1 A Genetic Algorithm

A genetic algorithm, also known as an evolutionary algorithm (Ahmed and Hermadi 2008; Michael et al. 2001; Mitchell 1998; Sivanandam and Deepa 2008), mimics natural selection via a randomized parallel search. Techniques based on genetic algorithms have been used to solve problems in the areas of artificial intelligence, machine learning, and function optimization. An initial set of possible solutions is randomly or manually generated (in our framework, a solution represents one test case). A fitness value of each possible solution with respect to an objective function is computed. The higher the fitness value that a possible solution has, the closer it is to the desired solution. Additional possible solutions can be generated based on existing possible solutions with higher fitness values; then, the fitness values of all possible solutions are updated. This process continues until a pre-defined stopping criterion is satisfied.

Two different evolutionary operators, *crossover* and *mutation*, can be applied in a GA to generate new possible solutions (Ahmed 2010). The former interchanges values between two existing possible solutions, while the latter maintains genetic diversity among different possible solutions. Therefore, in our proposed framework, we apply both operators at the same time to generate new test cases. More details on how to use the framework to generate new test cases can be found from the running example given in Section 3 and Section 4.

2.2 Software Product Line

One of the most prominent approaches in software engineering is Software Product Line (SPL) engineering, which aims to achieve systematic redeployment within the development of similar software products. Domains such as the automotive industry have already applied SPL engineering and have not only successfully ensured the quality of individual products but have also reduced the cost of developing new products. As the number of product variations rises, the number of component combinations likewise increases, resulting in millions of different configurations. For this reason, the benefit of applying SPL to software development is that a foundational set of assets can be used as the basis for building a variety of products (Kakarontzas et al. 2008).

During the development of a family of software products, SPL focuses on the extraction of the product variabilities as well as the commonalities, which represent a set of common core assets. Development for reuse as well as development with reuse can be identified as two key stages of SPL (Mohamed Ali and Moawad 2010). The product line infrastructure constructed in the former stage contains the assets that can be reused in the latter.

Testing for SPL aim to reduce the overall time and cost to guarantee the quality of the family of software under test. Due to the two distinct stages of SPL development, testing SPL differs from that of traditional software. Fully testing an SPL requires all possible product variants to be tested, which is not only inefficient but also ineffective. However, as we mentioned in Section 1, most of the techniques for testing SPL are still performed individually with respect to each product. Since "reuse" is the key point in SPL development, the test resource used for one product should also be "reused" when testing a new product in the same family. Such idea of "reuse" should be implemented in the testing for SPL, which is also the main contribution of this paper.

2.3 Structure-Based Coverage Criteria

In our study, we assess the effectiveness of the proposed framework by applying two categories of structure-based criteria − *control-flow* and *data-flow* − to investigate execution of certain structural components. Control-flow criteria examine the execution of logical expressions that establish flow of control. Two types of control-flow criteria include statement coverage and condition coverage. Statement coverage is fulfilled when every feasible statement in the program has been executed at least once. Condition coverage, which subsumes statement coverage, requires the invocation of both the true and false branches of every feasible condition.

In contrast, data-flow criteria evaluate execution through defined data (def) and their use (use) within the program. Def refers to the specific line of code containing a certain value assigned to a variable while use refers to the line of code in which the variable is used. Within the use criteria, c-use represents usage of a datum in a computational operation or as output of a function, while p-use represents usage of a datum in a predicate. All-def-use coverage (all-use coverage) uses both c-use and p-use in the investigation.

The coverage percentage that a test set achieves can be calculated as:

$$\text{Coverage Percentage} = \frac{N_c}{N_e - N_i} \tag{1}$$

In Eq. (1), N_c represents the number of certain structural components (statements, true/false branches, all-use pairs in this paper) covered (executed)[1] by a test set, N_e represents the total number of certain structural components in a program, and N_i represents the number of infeasible structural components. In this paper, we determine the value of N_i as the number of certain structural components that are not executed by a test set with 100,000 randomly generated test cases. A similar approach is used in (Chen et al. 2013).

2.4 Fault Localization Techniques

Fault localization techniques calculate the suspiciousness of program components (i.e. statement) based on a coverage matrix and a result vector. Assume that we have a program P which

[1] In the rest of this paper, "a structural component is *covered*" and "a structural component is *executed*" are equivalent.

contains n statements and a test set T with m test cases. With respect to the coverage matrix, each row represents a program component while each column represents a test case. Each entry c_{ij} in the matrix is 1 if the ith component is covered by the jth test case; otherwise, c_{ij} equals 0. The result vector is also binary such that each entry r_i is 1 if the ith test case results in failure, and 0 if it does not.

Of the fault localization techniques in the published literature, we select eight techniques for our case studies based on the following criteria: the techniques should have been demonstrated to be effective by other publications; the amount of data required by each technique is the same to avoid the possible unfairness.

Assume that a program P contains n statements.[2] Let s be a statement of P. Also, a test set T with m test cases has been executed against program P. Notations used in this section are listed as below:

N_{CF} number of failed test cases that cover the statement
N_{UF} number of failed test cases that do not cover the statement
N_{CS} number of successful test cases that cover the statement
N_{US} number of successful test cases that do not cover the statement
N_C total number of test cases that cover the statement
N_U total number of test cases that do not cover the statement
N_S total number of successful test cases
N_F total number of failed test cases
t_f a failed test case
t_i a test case in T

Tarantula (Jones and Harrold 2005), which is a simple yet effective technique in many cases, roots in the intuition that statements mainly covered by failed test cases are more likely to contain faults than those covered by successful test cases. The suspiciousness of s can be computed using the following formula:

$$susp(s) = \frac{N_{CF}(s)}{N_F} \Big/ \left(\frac{N_{CF}(s)}{N_F} + \frac{N_{CS}(s)}{N_S} \right) \tag{2}$$

As a more effective fault localization technique than Tarantula (Abreu et al. 2009), Ochiai calculates the suspiciousness of s as:

$$susp(s) = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}} \tag{3}$$

O and O^P are two fault localization techniques defined in (Naish et al. 2011). Naish et al. suggested that for single-bug programs, O shows better performance while O^P is better in dealing with programs with multiple bugs. With respect to programs containing exactly one bug, the faulty statement's N_{UF} remains 0 the entire time. As a result, those with a positive N_{UF} are not faulty; suspiciousness of any other statement is proportional to its N_{US}. The O technique defines the suspiciousness of s as:

$$susp(s) = \begin{cases} -1, & \text{if } N_{UF} > 0 \\ N_{US}, & \text{otherwise} \end{cases} \tag{4}$$

[2] Without loss of generality, program components are considered to be statements for the rest of the paper.

With respect to programs with multiple bugs, O^P shows better performance since it pushes the statements with larger N_{CF} and smaller N_{CS} to the top of the ranking using the following formula:

$$susp(s) = N_{CF} - \frac{N_{CS}}{N_S + 1} \tag{5}$$

Cross tabulation (Everitt 1977; Freeman 1987; Goodman and Duncan 1984), a.k.a. crosstab, is a method that studies the relationship among several categorical variables. A fault localization technique based on crosstab is proposed in (Wong et al. 2012b). By constructing two column-wise categorical variables (*covered* and *not covered*) and two row-wise categorical variables (*successful* and *failed*), crosstabs aid in analyzing the *association* degree between the failed (or successful) execution result and the coverage of each statement with null hypothesis that they are independent given beforehand. The degree of association will then be utilized to calculate the suspiciousness value for each statement.

H3b and H3c are proposed in (Wong et al. 2010) with the focus on how additional failed (or successful) test cases help locate faults. It is concluded that the contributions of failed (successful) test cases sequentially decrease. In other words, the first failed test case contributes greater than or equal to the second, while the second contributes greater than or equal to the third, and so on. The suspiciousness of each statement is calculated as $[(1.0) \times n_{F,1} + (0.1) \times n_{F,2} + (0.01) \times n_{F,3}]–[(1.0) \times n_{s,1} + (0.1) \times n_{s,2} + \alpha \times \chi_{F/S} \times n_{s,3}]$, where $n_{F,i}$ and $n_{S,i}$ are the number of failed and successful tests in the ith group, and $\chi_{F/S}$ is the ratio of the total number of failed to the total number of successful tests with respect to a given fault. The technique is named H3b when $\alpha = 0.001$ and H3c with $\alpha = 0.0001$.

Wong et al. proposed DStar (D*) (Wong et al. 2012a, 2014) based on Kulczynski coefficient. The suspiciousness of each statement is computed as:

$$susp(s) = \frac{(N_{CF}(s))^*}{N_{UF}(s) + N_{CS}(s)} \tag{6}$$

The effectiveness of D* increases as * grows before * exceeds a critical value. Since this paper is not focusing on the effectiveness of a specific fault localization technique, we set * as 3 and the technique is denoted as D^3 for the rest of the paper.

3 GA-Based Test Generation for SPL

In this section, we demonstrate the framework of the proposed GA-based test generation for SPL. In Section 3.1, we describe the mechanism by which test cases are converted to binary forms. Next, in Section 3.2, we present the detailed procedure used to apply our framework to generate a test set for a product of an SPL, which includes two approaches: test generation using Euclidean distance (EC) as well as using K-Means clustering (KM). Finally, Section 3.3 explores how to apply the test sets generated for one product to future products.

Figure 1 illustrates the basic procedure of our framework. The cycle marked with blue arrows represents the test generation process for one product in a specific software product line (Section 3.2), while the orange cycle displays how test cases generated for one product can be reused to test another product of the same product family (Section 3.3). In addition, a detailed process of how new test cases are generated based on the current test set is demonstrated with the dashed green cycle. Two approaches, EC (Section 3.2.1) and KM (Section 3.2.2) have been

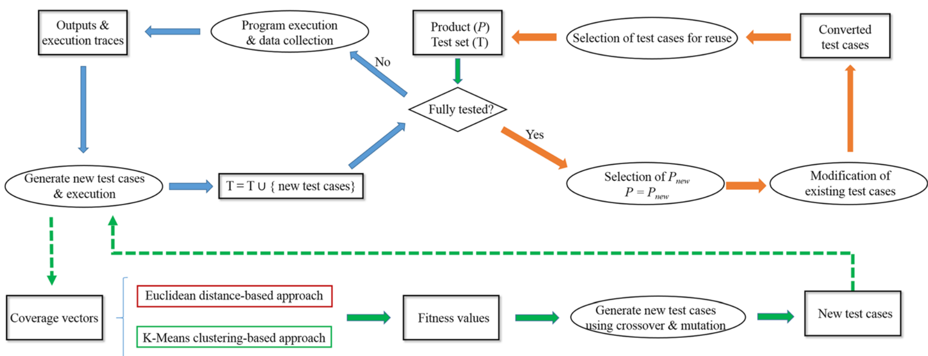


Fig. 1 Procedure of GA-based test generation for SPL

proposed to calculate the fitness values of each test case. These values help decide to which test cases the genetic operators (*crossover* and *mutation*) should be applied to generate new ones.

3.1 Encoding of Test Cases

Before a genetic algorithm can guide the test generation process, a mechanism should be established to convert each test case to a unique binary string.[3] In this binary string, each digit is either 0 or 1 such that the evolutionary operators (*crossover* and *mutation*) discussed in Section 2.1 can create new test cases by modifying existing test cases. Below, we introduce how to convert different primitive data types (i.e. *Integer*, *Floating point number*, *Boolean*, *Character*, *String*, and *Enumeration*) of input parameters to binary form.

- *Integer*:

  Each *Integer* can easily be presented as a binary string. For example, $(60)_{10} = (111100)_2$. However, in order to limit the search space of GA, sometimes we need to define the maximum (or minimum) possible value for a specific *Integer* parameter. This also helps us define the maximum number of digits used to represent this parameter.

  For example, with respect to an *Integer* input parameter α, the maximum possible value is 100 while the minimum is −100. As a result, the parameter can be converted to an 8-bit string as follows: since $(100)_{10} = (1100100)_2$, seven bits are required to represent its absolute value[4]; the eighth bit shows whether the value is positive (if the bit is 1) or negative (if the bit is 0).

  Suppose that the input parameter $\alpha = 60$. Based on our previous description, the value will be converted to {01111001}:

$$\underbrace{0111100}_{60\ \text{in binary form}}\qquad\underbrace{1}_{\text{positive}}$$

  Note also that during the test generation process, invalid input values are likely to be created. For example, based on the assumption above, string {11101001} which represents

[3] Several other encoding approaches exist, such as permutation encoding, value encoding, and tree encoding. In this paper, we focus on binary encoding.

[4] If the absolute values of the upper bound and lower bound differ, we choose the one with greater absolute value to determine the number of bits needed.

$(116)_{10}$ can also be produced. In order to simplify the test generation process, we abandon these productions and generate new test cases until they satisfy the specifications.

- *Floating point number:*

  With respect to input parameters that accept *Floating point numbers*, we divide the number into two parts: the integer part and the decimal part. Each part can be presented as an individual integer. Similar to *Integer*, an additional bit is required to represent whether it is positive or negative. For instance, with respect to $(11.11)_{10}$, the integer part is converted as $(1011)_2$ while the decimal part is $(0001011)_2$ because we need at least seven bits to represent $(.99)_{10}$ as $(99)_{10} = (1100011)_2$.[5] As a result, $(11.11)_{10}$ is converted as:

$$\underbrace{1011}_{\text{integer}}\ \underbrace{0001011}_{\text{decimal}}\ \underbrace{1}_{\text{positive}}$$

- *Boolean:*

  If an input parameter accepts *Boolean* data type, we use "1" for "True" and "0" for "False".
- *Character:*

  Each *Character* will be converted to its corresponding ASCII code.
- *String:*

  A *String* can be presented as a sequence of characters. Similar to the encoding of *Integer* type, we define the maximum number of characters with respect to a specific *String* type input parameter. As a result, strings can be converted to a sequence of ASCII codes.
- *Enumeration:*

  Each possible value of an *Enumeration* type can be presented as a unique binary string. Assume that input parameter β accepts four possible values, "A", "B", "C", and "D". Table 1 can be used to convert a test case into binary form.

Under certain circumstances, the number of possible values for a parameter may not be 2^n, where n represents the number of bits needed to represent the parameter. In these cases, binary codes assigned to each input value can be modified based on the experience of testers or the operational profiles. Now assume the input parameter β accepts only three possible values, "A", "B", and "C". Assume also that the function corresponding to "C" is used more often than those corresponding to "A" and "B". For this reason, the "C" function should be tested more thoroughly than the "A" and "B" functions. Table 2 contains the updated input values for β.

For other existing data types (especially composite data types such as *Time*, *List*, and *Graph*), we first divide the data into the above primitive data types. This implies that a complex data type can be represented by several primitive types. For example, with respect to *Time*, it is divided into six *Integer* data (year, month, day, hour, minute, and second). In addition, with respect to a specific data structure that also contains structural information (such as array, list, graph, etc.), we keep a complete record of the sequence of all the data within the structured data element to prevent any loss of information.

Consider the following scenario. Assume an array is used as a test case for a program *P*. The values of three integer input parameters, X, Y, and Z, are stored in this array. Without loss of generality, we assume that 10 bits are needed for each input parameter (as discussed above). Instead of only transforming the values of X, Y, and Z into binary strings, we also record the

[5] We assume that the upper bound and lower bound of the parameter are 15.00 and −15.00, respectively. Since $(15)_{10} = (1111)_2$, we only need four bits to represent the integer part of the parameter.

Table 1 Example for *enumeration* type

Input values for β	Binary string
A	00
B	01
C	10
D	11

index of these three parameters and their offsets. As shown in Table 3, the input parameter X has an index 0, implying that it is the first parameter in the input array. Its offset is 10, indicating that this parameter needs 10 bits. The same applies for Y and Z.

With respect to other data structures such as graph, tree, and linked list, we also store the relationships between different input parameters. By doing so, we can assure that various input values of a complex data type can be represented as a binary string without any information losses.

3.2 Two GA-Based Test Generation Approaches

Let us assume that we need to generate a test set *T* for a program (say *P* with *n* lines of code) of an SPL. Without loss of generality, suppose the generation stops if *T* achieves certain statement coverage on *P*. Other stopping criteria, such as a maximum number of test cases for *T*, can also be used. The following provides the necessary steps:

1) Initialization of *T*. If there are test cases from other products to be reused, *T* is initialized to contain these test cases. Otherwise, *T* has a small number (say *k*) of randomly generate test cases for *P*.
2) Coverage measurement of each test case in *T*. Let $c_i = (c_{i1}, c_{i2}, c_{i3}, c_{i4}, \ldots, c_{in})$, $i \le k$, be the statement coverage vector of the *i*th test case, where c_{ij} ($j \le n$) has a value of 1 or 0, indicating whether the *j*th statement is covered by the *i*th test case or not. If $c_{ij} = 1$, it is covered; otherwise, it is not.
3) Computation of a fitness value f_i for each test case. This value can be computed using two approaches, EC and KM, discussed later.
4) Generation of new test cases. The proportional roulette wheel selection (Mitchell 1998) is applied to determine which test case(s) in *T* will be used to generate new test cases. The probability of the *i*th test case being chosen is $f_i / \sum_{i=1}^{k} f_i$. *Crossover* and *mutation* operators are then applied on the chosen test cases to get two new test cases. The probability of each bit being mutated (mutation rate) is 0.30. If the newly generated test cases already exist in *T*, repeat step (4) until we have a predefined number of test cases not in *T*.
5) Addition of new tests to *T*. New test cases generated at step (4) are incorporated into *T*. The cumulative statement coverage on *P* achieved by all the tests in *T* is measured.
6) Coverage measurement of newly generated test cases. Similar to step (2), the statement coverage vectors of the new test cases are collected.

Table 2 Example for *enumeration* type with less than 2^n (in this case $n = 2$) possible values

Input values for β	Binary string
A	00
B	01
C	10,11

Table 3 Record for sequence information

Input parameter	Index	Offset
X	0	10
Y	1	10
Z	2	10

7) Decision on whether test generation should continue. If the predefined statement coverage is satisfied, test generation stops; otherwise, the next iteration begins from step (3).

With respect to step (3), two approaches, EC and KM, are presented in Section 3.2.1 and Section 3.2.2 to calculate the fitness value of each test case.

For step (4), the efficiency of test generation process could vary due to the number of test cases generated in each iteration, especially for KM approach. As a result, we apply the following method to improve its efficiency. If T contains fewer than 100 test cases, two test cases are generated in each iteration; If T contains more than 100 test cases but less than 500 test cases, 5 test cases are generated in each iteration; In addition, if T contains more than 500 test cases, 20 test cases are generated in each iteration.

3.2.1 Using the Euclidean Distance (EC) to Calculate the Fitness Value of Each Test Case

In mathematics, the Euclidean distance (or metric) is the "ordinary" distance (such as a straight line) between two data points in Euclidean space. Assume we have two data points, $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$, in Cartesian coordinates. Then the Euclidean distance between p and q is:

$$EC(p, q) = \|p - q\| = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{7}$$

Intuitively speaking, if the Euclidean distance between the coverage vectors of two test cases is *short*, it suggests that the two test cases are likely to cover similar parts of the program. On the other hand, if the distance is *large*, the two test cases could cover different parts of the program. As a result, test cases with larger Euclidean distances tend to have different execution traces when compared to those with shorter distances. This also implies that new test cases generated based on tests that have *larger* (instead of *shorter*) Euclidean distances with current tests will have a higher probability to cover statements that are not executed by current test cases. The threat to validity of this assumption will be discussed in Section 10.

With the understanding of Euclidean distance, we now explain how to compute the fitness value f_i for the ith test case. The following steps are employed to assign different fitness values to test cases: assume we have the coverage vectors $c_1, c_2 \ldots c_k$ for $t_1, t_2 \ldots t_k$. The fitness value of the ith test case in T is assigned using the steps listed below:

1) Computation of an objective vector $\overline{c}$ as

$$\overline{c} = \left(\frac{c_{11} + c_{21} + \cdots + c_{k1}}{k}, \frac{c_{12} + c_{22} + \cdots + c_{k2}}{k}, \ldots, \frac{c_{1n} + c_{2n} + \cdots + c_{kn}}{k} \right) \tag{8}$$

2) Compute the Euclidean distance between c_i and $\overline{c}$, $EC(c_i, \overline{c})$.

3) EC(c_i, $\bar{c}$) is assigned to the ith test case as its fitness value

$$f_i = EC\left(c_i, \bar{c}\right) \quad (9)$$

3.2.2 Using K-Means Clustering (KM) to Calculate the Fitness Value of Each Test Case

K-Means clustering is a popular and effective method for cluster analysis in data mining. It aims to partition data points based on their similarity. In other words, data points within the same cluster are more similar than those in other clusters. In order to have a better understanding of the proposed approach, this section provides a brief introduction to K-Means clustering and an example illustrating how it works.

The idea of K-Means clustering was first proposed by Stuart Lloyd in 1957 and published in 1965 by E. W. Forgy (1965). As a result, the method is also referred to as Lloyd-Forgy. In a 1967 paper, MacQueen (1967) coined the term "K-Means". The algorithm was improved by Hartigan and Wong in 1979 (1979).

Given a set of data points ($x_1, x_2, \ldots, x_n$), in which each data point is a d-dimension vector, K-Means clustering aims to group the n data points into K ($K \le n$) clusters $S = \{S_1, S_2, \ldots, S_k\}$ such that the following objective is fulfilled:

$$\underset{S}{\arg\min} \sum_{i=1}^{K} \sum_{x \in S_i} \|x - \mu_i\|^2 \quad (10)$$

where μ_i is the mean of data points in S_i.

An iterative optimization algorithm, which is referred to as a K-Means algorithm, can be used to determine the K centroids and corresponding clusters. The algorithm contains the following steps:

1) Determine K and randomly select K data points as initial centroids.
2) Include each data point into the cluster such that the squared Euclidean distance between the data point and the corresponding centroid is less than or equal to that between this point and any other centroids.
3) The updated centroids are:

$$c_i^{(j)} = \frac{1}{\left|S_i^{(j-1)}\right|} \sum_{x \in S_i^{(j-1)}} x \quad (11)$$

   where $c_i^{(j)}$ is the centroid of the ith cluster at the jth iteration and $|S_i^{(j-1)}|$ is the number of data points in the ith cluster at the (j-1)th iteration.
4) If the centroids remain unchanged between two consecutive iterations, the clustering procedure completes. If not, the next iteration begins at step (2).

A simple example is provided for a better understanding of the algorithm above, which is essential to the comprehension of the proposed approach using K-Means clustering to calculate the fitness value of each test case.

Four data points are listed in Table 4. Without loss of generality, we choose $K = 2$ and $c_1^{(0)} = x_1$, $c_2^{(0)} = x_2$ as the two initial centroids. See Fig. 2a, in which squares represent four data points while hollow diamonds represent the centroids.

Table 4 Data points for K-means clustering

Data point	Attribute 1	Attribute 2
x_1	1	1
x_2	2	1
x_3	4	3
x_4	5	4

Based on these initial values, as $c_1^{(0)} = x_1$ and $c_2^{(0)} = x_2$, x_1 will be assigned to the cluster of $c_1^{(0)}$ and x_2 to the cluster of $c_2^{(0)}$. For x_3, because the squared Euclidean distance between x_3 and $c_2^{(0)}$ is obviously less than that between x_3 and $c_1^{(0)}$, x_3 is assigned to the cluster of $c_2^{(0)}$. Similarly, x_4 is also assigned to the cluster of $c_2^{(0)}$. As a result, we have two clusters which are $S_1^{(0)} = (x_1)$ and $S_2^{(0)} = (x_2, x_3, x_4)$.

We then compute the updated centroids using Eq. (11):

$$c_1^{(1)} = \frac{1}{\left|S_1^{(0)}\right|} \sum_{x_i \in S_1^{(0)}} x_i = (1, 1)$$

$$c_2^{(1)} = \frac{1}{\left|S_2^{(0)}\right|} \sum_{x_i \in S_2^{(0)}} x_i = \left(\frac{11}{3}, \frac{8}{3}\right)$$

The updated centroids and four data points are displayed in Fig. 2b. Because the centroids changed in this iteration, the process must continue.

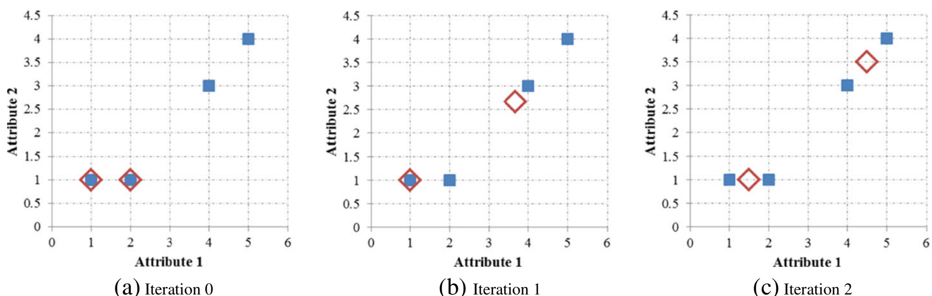
In Iteration 2, x_1 will be assigned to the cluster of $c_1^{(1)}$ since $c_1^{(1)} = x_1$. According to Fig. 2b, we can clearly identify that the squared Euclidean distance between x_2 and $c_1^{(1)}$ is less than that between x_2 and $c_2^{(1)}$. Therefore, x_2 will be assigned to the cluster of $c_1^{(1)}$. Following the same procedure, x_3 and x_4 are assigned to the cluster of $c_2^{(1)}$. The two clusters are $S_1^{(1)} = (x_1, x_2)$ and $S_2^{(1)} = (x_3, x_4)$.

The centroids are then updated again and illustrated in Fig. 2c

$$c_1^{(1)} = \frac{1}{\left|S_1^{(1)}\right|} \sum_{x_i \in S_1^{(1)}} x_i = \left(\frac{3}{2}, 1\right)$$

$$c_2^{(2)} = \frac{1}{\left|S_2^{(1)}\right|} \sum_{x_i \in S_2^{(1)}} x_i = \left(\frac{9}{2}, \frac{7}{2}\right)$$

If we proceed to the next iteration, we find the centroids and the clusters remain unchanged from Iteration 3 and conclude that the first cluster contains data points (x_1, x_2) and the second cluster has (x_3, x_4).



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Fig. 2 Three iterations for the example

The rationale behind the KM approach is that if the execution trace of one test case is similar to that of a failed test case, then the former is also likely to be a failed test and provides more hints on the bug location. As a result, if either a failed test case or a test case similar to a failed test is selected, it is more likely that new test cases generated can help us better determine the location of bugs in the program being tested. The potential threat to validity of this approach will be discussed again in Section 10. The same idea has also been used in a recently published paper (Gao et al. 2015).

With this introduction, we now explain the procedure to compute the fitness value f_i for the ith test case based on K-Means clustering. Assume we have the coverage vectors $c_1, c_2 ... c_k$ for $t_1, t_2 \ldots t_k$.

1) Determine K and the initial centroids. The following three scenarios should be considered:

a) If T contains only successful test cases, K equals 2 and the coverage vector c_r of a randomly selected test case t_r is assigned as $c_1^{(0)}$. Identify a test case t_p such that $EC(c_p, c_r)$ is not less than $EC(c_i, c_r), \forall t_i \in T$. c_p is assigned as $c_2^{(0)}$;
b) If only one failed test case t_f is in T, K equals 2 and the coverage vector c_f of t_f is assigned as $c_1^{(0)}$. Identify a test case t_l such that $EC(c_f, c_l)$ is not less than $EC(c_f, c_i), \forall t_i \in T$. c_l is assigned as $c_2^{(0)}$;
c) If in T there exist two or more failed test cases, the steps listed below are employed to determine K and the initial centroids. Assume in T we have λ failed test cases denoted as $(t_{f1}, t_{f1}, \ldots, t_{f\lambda})$; consequently, $(t_{s1}, t_{s2}, \ldots, t_{s(k-\lambda)})$ represents all the successful test cases.

i. Compute the Euclidean distances between the coverage vector of a failed test case and that of any successful test case $EC(t_{fi}, t_{sj}), 0 < i \leq \lambda, 0 < j \leq n - \lambda$;
ii. Set a similarity threshold $\theta = \alpha \times \max_{\mathrm{all}(i,j)} EC(t_{fi}, t_{sj})$ where α is a constant between 0 and 1. For the rest of the paper we use $\alpha = 0.1$ unless otherwise stated;
iii. Compute the Euclidean distance between any pair of the coverage vectors of failed test cases. For $0 < i < j \leq \lambda$, if $EC(t_{fi}, t_{fj}) < \theta$, these two failed test cases belong to the same group, otherwise, they belong to different groups;
iv. Assume after step (iii), we have ω groups of failed test cases. K equals ω, and we randomly select one coverage vector from each group of failed test cases as the initial centroids. However, if $\omega = 1$, then K equals 2 instead.

2) Apply K-Means clustering to partition all the test cases.
3) Compute the Euclidean distance between each test case and its corresponding centroid. Assume for the rth iteration, the ith test case is partitioned into the jth cluster S_j and the coverage vectors of t_p and t_q have the largest and smallest Euclidean distance, respectively, to the corresponding centroid $c_j^{(r)}$ within the cluster. Assume also that N_j represents the number of data points in the jth cluster. Then the fitness value of the ith test case is:

$$f_i = \begin{cases} \dfrac{EC\left(c_p, c_j^{(r)}\right)}{EC\left(c_i, c_j^{(r)}\right)}, & w_i \neq c_j,\ N_j > 2 \\ \dfrac{EC\left(c_p, c_j^{(r)}\right)}{EC\left(c_q, c_j^{(r)}\right)}, & w_i = c_j, N_j > 2 \\ \xi, & w_i = c_j, N_j \leq 2 \end{cases} \tag{12}$$

With respect to the value of ξ, if the ith test case is successful, $\xi = 1$; otherwise, $\xi = \max_{t_v \in T, v \neq i} f_v, \delta > 0$.

We also include the discussion on the possible impact of α on the effectiveness of KM approach in generating test cases in Section 7.

3.3 Test Cases Reuse Based on Fault Localization Techniques

With the test cases generated for one product in the product family, the test cases can be reused to test another product in that family. In software product line engineering, the essential concept is "reuse", which means that the development of different products within the same product family is based on the "core" components of a specific SPL. Furthermore, a significant number of modules/components, referred to as commonalities, are shared by different products. As a result, the same or similar bugs could reside in several products in a product family. Intuitively, fault localization techniques can be employed to determine which test cases should be reused. With the help of statement rankings generated by fault localization techniques, test cases which cover statements with higher suspiciousness values are more beneficial to testers and help debug the product with less effort.

Due to the difference between products, some of the test cases for one product may not be applicable for another. For example, suppose we have an SPL for drawing. One product contains functions to calculate the areas of five geometric figures, namely square, circle, triangle, rectangle, and ellipse. However, another product in the same family only provides the functions for four figures: square, circle, triangle, and rectangle. As a result, those test cases focusing on calculating the area of ellipse are not applicable in testing the new product and cannot be reused. A more detailed example is included in Section 4.

Suppose we have completed the test generation for one product P_1 and would like to reuse the test set T_1 with k test cases generated to test another product P_2. According to Section 3.2, the coverage matrix of T_1 consisting of k coverage vectors $(c_1, c_2, \ldots, c_k)$ are achieved from the test generation process. The following provides the necessary steps to determine which test cases should be reused:

1) Remove test cases in T_1 which are not applicable in T_2.
2) Compute the suspiciousness values of each statement in P_1.

   Fault localization techniques described in Section 2.4 (H3b, H3c, Crosstab, Tarantula, Ochiai, D^3, O and O^P) can be applied based on the coverage matrix and result vector.
3) Compute the sum of suspiciousnesses values of statements covered by each test case in T_1.

$$SumSusp(t_i) = \sum_{j=1}^{n} susp(s_j) \times c_{ij} \tag{13}$$

4) The reusability of each test case is computed by using $SumSusp(t_i)$ and f_i achieved in Section 3.2

$$RE(t_i) = \beta \times \frac{SumSusp(t_i)}{\max_{\text{All } i} SumSusp(t_i)} + \gamma \times \frac{f_i}{\max_{\text{All } i} f_i} \tag{14}$$

   where β and γ are constants and $\beta + \gamma = 1$. For the rest of the paper, $\beta = 0.5$ and $\gamma = 0.5$ unless clarified otherwise.

5) Arrange the test cases in a decreasing order based on the *RE* values. Two steps will be performed to select the test cases for reuse:

   i. Include those failed test cases in T_1 into the test set T_2 for P_2;
   ii. Select test cases from the remaining test cases in T_1 based on the ranking generated in step (5). Choose those with the highest *RE* values until the size of T_2 reaches the maximum number of test cases for reuse.

Similarly, the values of β and γ could have possible impact on the effectiveness of our reuse approach. As a result, sensitivity analyses are performed in Section 7 regarding this concern.

4 Running Examples

In this section, we go through the proposed framework with two sample programs, say P_1 and P_2, from an SPL for car insurance companies. In Section 4.1, P_1 with only one bug is used to illustrate how to generate test cases using EC or KM approach. In Section 4.2, the procedures of test case reuse as well as test generation for P_2 with multiple bugs are displayed.

4.1 Using EC and KM Approaches to Generate Test Cases

As shown in Fig. 3, P_1 helps employees of an insurance company to determine the insurance plans for various customers. It gives customers quotes based on the *price* (no more than \$100K,[6] e.g. \$45K) and *model* (compact, full, sports, or luxury) of their cars. There is a bug at s_5 which does not give users the correct insurance plan. A test set with two test cases t_1 ("Sports", "\$45K") and t_2 ("Luxury", "\$40K") is also provided as the initial test set. The black dot indicates that the corresponding statement is covered by a specific test case. The execution results of the two test cases are also included.

Before the two initial test cases can be used to generate more test cases, the encoding for these test cases must be done. Based on the description in Section 3.1, we use the first two bits to represent the parameter *model* (*Enumeration* type) and the following eight bits to represent *price* (*Integer* type). For the first two bits, {00} stands for "Compact", while {01}, {10}, and {11} stand for "Fullsize", "Sports", and "Luxury", respectively.

Take t_1 as an example. The value of *model* is represented by {10}, which means "Sports". For the parameter *price*, according to Section 3.1, as maximum value for *price* is 100, eight bits are required for each test case (seven bits for the numerical value and one bit for positive/negative). Since $(45)_{10} = (101101)_2$, we use the {01011011} to represent the value "\$45K" with the last bit "1" (positive) appended.

Therefore, the two test cases can be converted as:

- $t_1 = \{1001011011\} \rightarrow \{\text{"Sports"}, \text{"\$45K"}\}$
- $t_2 = \{1101010001\} \rightarrow \{\text{"Luxury"}, \text{"\$40K"}\}$

[6] For the purposes of illustration, we assume this is guaranteed by the input verification module, which is not included in the source code.

Stmt. #.	Program	Coverage t_1	Coverage t_2
s_1	read(model, price);	•	•
s_2	if ((model=="Compact" || model=="Fullsize") && price < 50)	•	•
s_3	type=Plan_A;		
s_4	else if (model=="Sports" && price < 50)	•	•
s_5	type = Plan_C; // Correct: type = Plan_B;	•	
s_6	else if (model=="Luxury" && price < 50)		•
s_7	type = Plan_C;		•
s_8	else if ((model=="Compact" || model=="Fullsize") && price >=50)		
s_9	type = Plan_D;		
s_{10}	else type = Plan_E;		
s_{11}	if (type != "Plan_D" && type != "Plan_E")	•	•
s_{12}	quote_price = 500 +price * 0.01 * 1000;	•	•
s_{13}	else if (type == "Plan_D")		
s_{14}	quote_price = 1000 +price * 0.01 * 1000;		
s_{15}	else quote_price = 1000 +price * 0.02 * 1000;		
s_{16}	print(type, quote_price);	•	•
Execution Results (1 =failed; 0=successful)		1	0

Fig. 3 A product with a single bug, a set of 2 test cases, and their execution traces

Suppose our goal is to generate a test set that achieves 100% statement coverage. By now, we only have two test cases. Since in each iteration we select two test cases to generate two more, t_1 and t_2 will be selected. Referring to step (4) in Section 3.2, the *crossover* operator will first be applied to the two test cases by swapping the bits of t_1 with those of t_2 between two randomly selected positions. In our case, bits between position 4 and position 6 are swapped, and we get the following two interim signatures:

- $$t_1^c = \{1101011011\}$$
- $$t_2^c = \{1001010001\}$$

The mutation operator is then applied by mutating its bits (1 to 0 or 0 to 1) at randomly selected positions. Without loss of generality, suppose bits of t_1^c at positions 1, 3, 4, 6, and 7 are mutated to generate a new test case t_3:

- $$t_3 = \{0110000011\} \rightarrow \{\text{“Fullsize”}\ \ \text{“\$65K”}\}$$

Similarly, by swapping five bits of t_2^c from positions 2 to 6, we can get another new test case t_4:

- $$t_4 = \{1110100001\} \rightarrow \{\text{“Luxury”},\ \ \text{“\$80K”}\}$$

After executing the two new test cases, our stopping criterion (100% statement coverage) is not fulfilled. As a result, the test generation process continues. Either EC or KM approach can be utilized to calculate the fitness values of the four test cases. We first use Eqs. (9) and (12) to calculate the fitness value of each test case shown in Table 5.

Among the four test cases, t_1 has the highest fitness value using either EC or KM, which means that its possibility of being chosen to generate new test cases is higher than the other three test cases. However, this does not indicate that test cases with

Table 5 Fitness values for t_1 to t_4 using EC and KM approaches

Test cases	EC fitness value	KM fitness value
t_1	1.69	2.11
t_2	1.19	1.00
t_3	0.69	1.11
t_4	1.09	1.11

low fitness values cannot be selected. Suppose that in this iteration, two more test cases are generated based on t_1 and t_3:

- $t_5 = \{0000101001\} \rightarrow \{\text{“Compact”}, \text{ “\$20K”}\}$
- $t_6 = \{0001111001\} \rightarrow \{\text{“Compact”}, \text{ “\$60K”}\}$

The updated fitness values of the six test cases are listed in Table 6. Also, the fitness values of the six test cases as well as the ranking of statements based on Ochiai are updated in Fig. 4.

4.2 Test Case Reuse

After completing the test generation for P_1, we move on to test P_2. While P_1 helps determine insurance plans for customers who own cars with a value of no more than \$100K, the highest car value accepted by P_2 is \$60K.[7] P_2 also requires an additional input parameter of *Boolean* data type, *isUsed*, which represents whether the car is used or brand new: customers with a used car will get a 10% discount on the quote price. The program is illustrated in Fig. 5.

We injected two faults into P_2. One fault is in s_5 and another in s_{17}. The former is the same as the fault existing in P_1 while the latter is newly injected. The reason is that similar bugs could exist in several products within an SPL since part of the source code in P_1 is reused in developing P_2. Also, implementing new functions could bring in new faults, such as the fault in statement s_{17}.

Due to the difference between P_1 and P_2, the encoding of test cases for P_2 are different from that for P_1. Now we only need seven bits instead of eight to represent the value of *price* since the maximum car value accepted by P_2 is 60.[8] For example, with respect to $t_1 = \{1001011011\}$, the last eight bits, $\{01011011\}$ will be converted to $\{1011011\}$ by deleting the first bit “0”.

Because P_2 requires another input parameter, *isUsed*, the binary strings for P_2 should include an additional bit: “1” represents a “used” car (“True” for *isUsed*) while “0” represents a “new” car (“False” for *isUsed*). As a result, a randomly generated bit is appended representing the value of *isUsed*. As a result, $t_1 = \{100101101\}$ will be converted to τ_1 as:

- $\tau_1 = \{1010110111\} \rightarrow \{\text{“Sports”}, \text{ “\$45K”}, \text{ “True”}\}$

An additional consideration is that not all the test cases for P_1 can be reused to test P_2. For example, with respect to t_3 and t_4, as the car price values in both test cases are more than 60, they are no longer applicable for P_2.

[7] For the purposes of illustration, we assume this is guaranteed by the input verification module, which is not included in the source code.

[8] $(111111)_2 = (63)_{10}$

Table 6 Fitness values for t_1 to t_6 using EC and KM approaches

Test cases	EC fitness value	KM fitness value
t_1	1.45	2.11
t_2	1.33	1.00
t_3	1.45	1.11
t_4	1.56	1.11
t_5	1.67	1.00
t_6	1.45	1.11

As a result, only four test cases, t_1, t_2, t_5, and t_6 from Section 4.1 can be reused to test P_2. These four test cases are converted based on the previous description and denoted as τ_1, τ_2, τ_3, and τ_4, respectively:

- $\tau_1 = \{1010110111\} \rightarrow \{\text{“Sports”}, \text{“\$45K”}, \text{“True”}\}$
- $\tau_2 = \{1110100011\} \rightarrow \{\text{“Luxury”}, \text{“\$40K”}, \text{“True”}\}$
- $\tau_3 = \{0001010011\} \rightarrow \{\text{“Compact”}, \text{“\$20K”}, \text{“True”}\}$
- $\tau_4 = \{0011110010\} \rightarrow \{\text{“Compact”}, \text{“\$60K”}, \text{“False”}\}$

Using Eq. (14), we can determine the reusability of each test case as listed in Table 7:

According to Table 7, the order of test cases based on EC Reusability is τ_1, τ_3, τ_2, and τ_4; when using KM Reusability, the order is τ_1, τ_2, τ_4, and τ_3. Referring to step (5) in Section 3.3, τ_1 is selected regardless of the approach used as τ_1 is a failed test case in the test generation for P_1. τ_2, τ_3, and τ_4 are chosen based on their respective ranking listed in Table 7 and the intended maximum number of test cases for reuse. For example, if the maximum number is 2, then only τ_1 and τ_3 would be selected using EC approach while τ_1 and τ_2 would be selected using KM approach.

<table>
<tr><th rowspan="2">Stmt. #.</th><th rowspan="2">Program</th><th colspan="6">Coverage</th><th colspan="2">Ochiai</th></tr>
<tr><th>t1</th><th>t2</th><th>t3</th><th>t4</th><th>t5</th><th>t6</th><th>Rank</th><th>susp</th></tr>
<tr><td>s1</td><td>read(model, price);</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>4</td><td>0.41</td></tr>
<tr><td>s2</td><td>if ((model=="Compact" || model=="Fullsize") && price < 50)</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>4</td><td>0.41</td></tr>
<tr><td>s3</td><td>type = Plan_A;</td><td></td><td></td><td></td><td></td><td>•</td><td></td><td>8</td><td>0</td></tr>
<tr><td>s4</td><td>else if (model=="Sports" && price < 50)</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td>3</td><td>0.45</td></tr>
<tr><td>s5</td><td>type = Plan_C; // Correct: type = Plan_B;</td><td>•</td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>0.71</td></tr>
<tr><td>s6</td><td>else if (model=="Luxury" && price < 50)</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td>8</td><td>0</td></tr>
<tr><td>s7</td><td>type = Plan_ C;</td><td></td><td>•</td><td></td><td></td><td></td><td></td><td>8</td><td>0</td></tr>
<tr><td>s8</td><td>else if ((model=="Compact" || model=="Fullsize") && price >= 50)</td><td></td><td></td><td>•</td><td>•</td><td></td><td>•</td><td>8</td><td>0</td></tr>
<tr><td>s9</td><td>type = Plan_ D;</td><td></td><td></td><td>•</td><td></td><td></td><td>•</td><td>8</td><td>0</td></tr>
<tr><td>s10</td><td>else type = Plan_E;</td><td></td><td></td><td></td><td>•</td><td></td><td></td><td>8</td><td>0</td></tr>
<tr><td>s11</td><td>if (type != "Plan_D" && type != "Plan_E")</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>4</td><td>0.41</td></tr>
<tr><td>s12</td><td>quote_price = 500 + price * 0.01 * 1000;</td><td>•</td><td>•</td><td></td><td></td><td>•</td><td></td><td>2</td><td>0.5</td></tr>
<tr><td>s13</td><td>else if (type == "Plan_D")</td><td></td><td></td><td>•</td><td></td><td></td><td>•</td><td>8</td><td>0</td></tr>
<tr><td>s14</td><td>quote_price = 1000 + price * 0.01 * 1000;</td><td></td><td></td><td>•</td><td></td><td></td><td>•</td><td>8</td><td>0</td></tr>
<tr><td>s15</td><td>else quote_price = 1000 + price * 0.02 * 1000;</td><td></td><td></td><td></td><td>•</td><td></td><td></td><td>8</td><td>0</td></tr>
<tr><td>s16</td><td>print(type, quote_price);</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>4</td><td>0.41</td></tr>
<tr><td colspan="2">Execution Results (1=failed; 0= successful)</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td><td></td></tr>
<tr><td colspan="2">EC Fitness Value</td><td>1.45</td><td>1.33</td><td>1.45</td><td>1.56</td><td>1.67</td><td>1.45</td><td></td><td></td></tr>
<tr><td colspan="2">KM Fitness Value</td><td>2.11</td><td>1.00</td><td>1.11</td><td>1.11</td><td>1.00</td><td>1.11</td><td></td><td></td></tr>
</table>

Fig. 4 Suspiciousness of each statement in the product, as well as the execution traces and fitness values of each test case

Stmt. #.	Program	Coverage τ_1	τ_2	τ_3	τ_4
s_1	read(model, price, isUsed);	•	•	•	•
s_2	if ((model=="Compact" || model=="Fullsize") && price < 50)	•	•	•	•
s_3	type = Plan_A;			•	
s_4	else if (model=="Sports" && price <50)	•	•		•
s_5	type = Plan_C; // Correct: type = Plan_B;	•			
s_6	else if (model=="Luxury" && price < 50)		•		•
s_7	type = Plan_C;		•		
s_8	else if ((model=="Compact" || model=="Fullsize") && price >=50)				•
s_9	type = Plan_D;				•
s_{10}	else type = Plan_E;				
s_{11}	if (type != "Plan_D" && type != "Plan_E")	•	•	•	•
s_{12}	quote_price = 500 +price * 0.01 * 1000;	•	•	•	
s_{13}	else if (type == "Plan_D")				•
s_{14}	quote_price = 1000 +price * 0.01 * 1000;				•
s_{15}	else quote_price = 1000 +price * 0.02 * 1000;				
s_{16}	if (isUsed)	•	•	•	•
s_{17}	quote_price = quote_price * 0.8; // Correct: quote_price = quote_price * 0.9;				•
s_{18}	print(type, quote_price);	•	•	•	•
Execution Results (1=failed; 0=successful)		1	0	0	1
EC Reusability ($\beta=0.5, \gamma=0.5$)		1.87	1.58	1.65	1.50
KM Reusability ($\beta=0.5, \gamma=0.5$)		2.00	1.25	1.12	1.16

Fig. 5 A product with two bugs, as well as the execution traces and fitness values of 4 test cases

Without loss of generality, suppose the maximum number of test cases for reuse is 4. In this scenario, all four test cases are included in the initial test set T_2 for P_2. Figure 5 shows the coverage results following execution of the test cases.

Assume the stopping criterion is the same as that in Section 4.1, which is 100% statement coverage. Following the same procedure, we can generate two more test cases:

- $\tau_5 = \{1110011111\} \rightarrow \{\text{“Luxury”}, \text{“\$55K”}, \text{“True”}\}$
- $\tau_6 = \{1001100110\} \rightarrow \{\text{“Sports”}, \text{“\$25K”}, \text{“False”}\}$

The fitness values using EC and KM approaches as well as the ranking based on Ochiai is updated in Fig. 6.

5 Experimental Setup

In this section, we provide the experimental setup of our case studies. Section 5.1 includes a brief introduction of the four SPLs as well as all the variants of each SPL used in this paper. The data collection and evaluation metrics are presented in Sections 5.2 and 5.3, respectively.

Table 7 Reusability of τ_1 to τ_4

Test case	*SumSusp*	EC reusability	KM reusability
$\tau_1 = \{1010110110\}$	3.30	1.87	2.00
$\tau_2 = \{1110100010\}$	2.59	1.58	1.25
$\tau_3 = \{0001010010\}$	2.14	1.65	1.12
$\tau_4 = \{0011110011\}$	2.09	1.50	1.16

		Coverage						Ochiai	
Stmt. #.	Program	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	Rank	susp
s_1	read(model, price, isUsed);	•	•	•	•	•	•	5	0.707
s_2	if ((model=="Compact" || model=="Fullsize") && price < 50)	•	•	•	•	•	•	5	0.707
s_3	type = Plan_A;			•				15	0
s_4	else if (model=="Sports" && price < 50)	•	•		•	•	•	3	0.775
s_5	type = Plan_C; // Correct: type = Plan_B;	•					•	1	0.820
s_6	else if (model=="Luxury" && price < 50)		•		•	•		13	0.408
s_7	type = Plan_C;		•					15	0
s_8	else if ((model=="Compact" || model=="Fullsize") && price >= 50)				•	•		13	0.408
s_9	type = Plan_D;				•			9	0.577
s_{10}	else type = Plan_E;					•		15	0
s_{11}	if (type != "Plan_D" && type != "Plan_E")	•	•	•	•		•	3	0.775
s_{12}	quote_price = 500 + price * 0.01 * 1000;	•	•	•			•	9	0.577
s_{13}	else if (type == "Plan_D")				•			9	0.577
s_{14}	quote_price = 1000 + price * 0.01 * 1000;				•			9	0.577
s_{15}	else quote_price = 1000 + price * 0.02 * 1000;					•		15	0
s_{16}	if (isUsed)	•	•	•	•	•	•	5	0.707
s_{17}	quote_price = quote_price * 0.8; // Correct: quote_price = quote_price * 0.9;				•		•	2	0.816
s_{18}	print(type, quote_price);	•	•	•	•	•	•	5	0.707
Execution Results (1=failed; 0=successful)		1	0	0	1	0	1		
EC Fitness Value		1.13	1.65	1.98	2.43	2.46	2.02		
KM Fitness Value		3.36	1.28	1.53	3.58	3.25	3.38		

Fig. 6 Suspiciousness of each statement in the product, as well as the execution traces and fitness values of each test case

5.1 Subject Programs

For our experiments, we applied the proposed framework as well as RT on four different product lines:

- *Photo Editing Line* (*PEL*) is an SPL for photo editing, which performs various photo editing tasks, such as photo batch processing, skin smoothing, advanced color adjustment, etc. Each product within PEL contains more than 3 K lines of C code.
- *E-Mail* (*EM*) is based on the email system model developed by Hall (Hall 2015). In addition to the original four features, we added seven to make the system more practical. The least lines of code a product within EM has is 1,094, while the most contains more than 2 K.
- *Public Health Complaint* (*PHC*) is implemented based on the model provided by Lee et al. (Lee et al. 2002). The SPL contains features such as food complaint, animal complaint, drug complaint, etc. Every product in PHC has more than 2 K lines of C code.
- *JFreeChart* (*JFC*) is a library that supports the display of a rich set of charts. The original library contains 91,174 lines of Java code. We modified the source code to get four unique products with different features.

The detailed information of the subject programs is included in Table 8.

For the rest of the paper, products within *PEL* are denoted as P_1, P_2, P_3, P_4, P_5, and P_6, respectively. Similarly, E_1 to E_6 are used to represent the six products within *EM*, H_1 to H_6 for *PHC*, and J_1 to J_4 for *JFC*.

Table 8 Subject programs

SPLs	Programming language	Lines of code
Photo Editing Line	C	3,087 ~ 3,589
E-Mail	C	1,094 ~ 2,143
Public Health Complaint	C	2,056 ~ 2,378
JFreeChart	Java	62,485 ~ 91,174

In this paper, we also created faulty versions of each product by applying mutation-based fault injection. Studies such as (Andrews et al. 2005; Do and Rothermel 2006; Liu et al. 2006; Namin et al. 2006) have shown that mutation-based fault injection is an effective simulation of real faults and provides trustworthy experimental results. The following two sets of mutation operators are employed in generating faulty versions:

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

Studies, including (Offutt et al. 1996; Wong and Mathur 1995a, b) cited in this paper, have shown that test cases with the capacity to kill mutants generated by replacement of a relational or logical operator, the possibility that they kill other mutants is high. Moreover, these test cases also achieve high code coverage. Although the focus of this paper differs from these studies, the conclusions can still be applied here.

5.2 Data Collection

For each product (P_1 to P_6, E_1 to E_6, H_1 to H_6, and J_1 to J_4), we use our proposed framework as well as random test generation (RT) to generate multiple test sets with respect to a fixed size or a fixed coverage (statement coverage, condition coverage, and all-use coverage (Mathur 2008)). Note that RT randomly generates test cases based on the same specifications used by EC and KM to ensure the comparison between RT and EC/KM is fair. For a fixed size, test sets of size 2, 3 … 10, 20 … 100, 200 … 1000, respectively, are generated.[9] On the other hand, test sets that achieve 85%, 90%, and 95% of all three types of coverage are generated for a fixed coverage.[10]

Multiple test sets are necessary because a large number of test sets will likely satisfy a given size or a given coverage achievement for each product. Therefore, selecting only one of these may lead to false conclusions. To alleviate this concern, we apply the following steps to ensure the statistical stability of the observed values (coverage achievements or number of test cases that achieve a specific coverage threshold):

1) With respect to a specific stopping criterion for test generation (i.e. 100 test cases, 70% statement coverage), we repeat the test generation procedure using EC, KM, or RT for a predefined number of times (say N_p) and achieve N_p independent test sets. In this paper, we set $N_p = 30$;
2) According to central limit theorem (Tijms 2004), we use Eq. (15) to calculate the estimated number of repetitions N needed to ensure that the observed values (i.e. coverage achieved by test sets of fixed size, number of test cases to get a predefined coverage) are statistically stable within a predefined confidence level of $(1 - \alpha) \times 100\%$ and accuracy range of $\pm r\%$

$$N = \left(\frac{100 \times \Phi^{-1}\left(\frac{2-\alpha}{2}\right) \times \sigma}{r \times \mu} \right)^2 \tag{15}$$

[9] For any general GA-based technique, at least two initial test cases are needed to generate new test cases. As a result, the minimum size of test sets is 2.

[10] For JFreeChart, we only include statement coverage and condition coverage due to the lack of tools in collecting data regarding all-use coverage.

where μ and σ represent the mean value and the standard deviation of the observed values, respectively. In the rest of the paper, we set $\alpha = 0.05$ and $r = 5$.

3) If $N > N_p$, the number of test sets is statistically reliable; Otherwise, generate $(N - N_p)$ test sets, let $N_p = N$, and return to step (2) to update N.

In this study, the coverage collection for C programs is supported by χSuds tool suite developed by Telcordia Technologies (Bellcore 1998). With respect to control-flow coverage, we employed χSuds to measure the "block coverage", which is equivalent to "statement coverage", and "decision coverage", which is equivalent to "condition coverage". As for data-flow coverage, "all-use coverage", which subsumes both c-use and p-use, is quantified. For Java programs, we use CodeCover (2016) to measure the coverage of test cases.

5.3 Evaluation Metrics

In this paper, we evaluate the effectiveness of the proposed framework using the following metrics. X and Y represent two test generation techniques.

- Average coverage achievement

  With respect to a test set size, if test sets with the same size generated by X achieve generally higher coverage than those generated by Y, then X is more effective than Y. For example, suppose that test sets generated by EC with a size of 10 achieve 60% statement coverage on average while those generated by RT only achieve 55%. Given these results, EC is considered to be better than RT (as relatively higher coverage can be achieved with same number of test cases).
- Average test set size

  With respect to a predefined coverage achievement (i.e. 85% statement coverage), X is more effective than Y if test sets generated by X are of smaller sizes than those generated by Y. For example, 85% statement coverage is fulfilled with 50 test cases generated by EC on average, while the number is 60 if using RT, therefore EC is more effective than RT (as fewer test cases are needed to achieve specific coverage achievements).
- The *EXAM* score

  The *EXAM* score (Wong et al. 2010, 2012b, 2007; Xie et al. 2013b) represents the percentage of statements within a program that need to be examined to locate the first bug. The higher EXAM score a technique has, the less effective it is.
- Average number of statements examined

  For test sets generated using a particular technique, the effectiveness of locating bugs can also be represented by calculating the average number of statements that need to be examined in order to locate the bug in every faulty version. Suppose the program P being debugged contains n faulty versions. This value can be computed as $\sum_{i=1}^{n} \chi(i)/n$ where $\chi(i)$ represents the number of statements that need to be examined to locate the bug within the ith faulty version of P.

  In the computation of average number of statements examined, we have to confess that in most of the cases multiple statements may share the same suspiciousness value. In other words, statements may be tied with same position in the ranking. In this paper, we apply two different level of effectiveness: best effectiveness (best case) and worst effectiveness (worst case). In the former, we examine the faulty statement first, while in the latter we examine all the statements without faults before examining the faulty statement.

6 Results

In this section, we demonstrate the experimental results of our case study. Section 6.1 shows the average coverage achievement of test sets generated by different approaches with respect to a fixed size. Average size of test sets with respect to fixed coverage is reported in Section 6.2. Section 6.3 presents the fault localization effectiveness using test sets generated by EC, KM, and RT (both programs with a single bug and with multiple bugs are considered).

6.1 Coverage Achievement With Fixed Test set Size

In this section, we examine the coverage achievements (statement, condition, and all-use) by test sets generated by EC, KM, and RT. The coverage achievements on the three products of *PEL*, P_1 to P_3, are illustrated from Fig. 7 through Fig. 9. The reason for only including these figures is purely in the interests of clarity and readability. In these figures, the horizontal axis represents the size of test sets in the logarithmic scale, while the vertical axis represents the average coverage percentage achieved by each approach.

For P_1 (Fig. 7), under each of the three coverage criteria, the average coverage achievement of test sets generated by EC is distinguishable from those generated by KM and RT. Take test sets with a size of 20 as an example, in Fig. 7a, test sets generated using EC achieve an average statement coverage of 93.01%, while the percentages for KM and RT are 90.01% and 90.03%, respectively. Of all the 84 scenarios (28 different test set sizes and three coverage criteria), KM exceeds EC in only five of them – test set size being 300, 400, 600, 700, and 800 with respect to all-use coverage. The largest difference between the coverages is 0.3%, in which case test sets with size of 300 by KM achieve an average of 97.80% all-use coverage and those by EC achieve 97.50%. In addition, RT performs better than EC in only one scenario: with test set size of 400, test sets by RT achieve an average of 98.11% all-use coverage, while those by EC achieve 98.00%. In summary, EC outperforms KM and RT significantly in generating test cases for P_1.

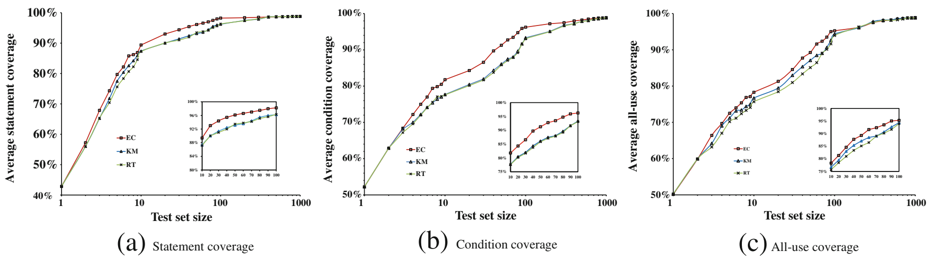
Also in Fig. 7, we observe that KM and *RT* are comparable in most of the scenarios. The average coverage achievements of test sets by KM and RT are identical. In Fig. 7a, for example, the average statement coverages for the test sets with 10 test cases are 87.38% with KM and 87.21% with RT. As the test set size increases, these values also increase and become 96.23% and 96.31%, respectively, for test sets of size 100. Similar results are observed in Fig. 7b and c.

When the test cases are reused (P_2 to P_3), however, the test sets generated by GA-based test generation approaches differ favorably from those generated by RT with respect to the average coverage achievement. For example, as seen in Fig. 8, RT achieves consistently lower coverage than the GA-based approaches. Similar results can also be observed in Fig. 9.

Differences can also be observed between EC and KM. The coverage achieved by KM for each of the three coverage criteria is generally lower than that achieved by EC. In Fig. 8, for example, KM underperforms EC for 92.9% (78 out of 84) of the experimental scenarios with respect to the three coverage criteria.

In addition to average coverage achievement, we also review detailed data of ten randomly selected test sets for each test set size. As shown in Fig. 10, for example, condition coverage for those ten test sets of size 60 for P_3 are presented.

In this figure, all test sets generated using GA-based approaches achieve higher coverage than those generated by RT with only one exception (test set 6, in which RT performs slightly better than



(a) Statement coverage (b) Condition coverage (c) All-use coverage

Fig. 7 Coverage achievements of GA-based test generation framework and RT on P_1

KM). Also, it can be observed that test sets generated using EC generally achieve higher statement coverage than those generated by KM, which is consistent with our findings in Fig. 7 through Fig. 9.

6.2 Test Set Size With Fixed Coverage

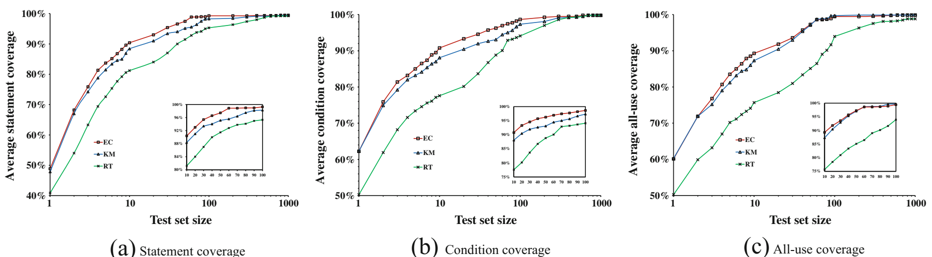
Table 9 presents the average number of test cases observed to reach a predefined coverage percentage for GA-based test generation approaches (EC and KM) and for RT. Instead of using P_1 to P_6, we present the results regarding J_1 to J_4 in order to make our observations more convincing. For example, based on Table 9, with respect to J_1, an average of 40.8 test cases achieves 85% statement coverage by using EC.

For each product, test sets generated by GA-based test generation approaches achieve the same level of coverage with smaller test sizes than those generated by RT. For each of the six scenarios (two coverage thresholds for each of three coverage criteria) for J_2 to J_4, the GA-based test generation approaches reach the given threshold with fewer test cases than RT. Similar results can also be observed in J_1 as well with only one exceptions: 90% condition coverage. The difference between RT and KM is less than 1%.

The impact of test case reuse between products is also significant. For example, with respect to J_1, EC requires an average of 40.8 test cases to achieve 85% statement coverage while RT needs 60.7. When it comes to J_2, the number for EC decreases to 25.7; however, RT still needs 67.4 test cases to reach the same level of statement coverage which increases the original value by 11%.

Differences can also be observed between EC and KM. For most of the scenarios (18 out of 24), EC has an advantage over KM, while KM performs better in the other 6 scenarios.

Figure 11 reports, for ten randomly selected test sets generated using different approaches, the number of test cases needed to achieve 85% statement coverage for J_3.



(a) Statement coverage (b) Condition coverage (c) All-use coverage

Fig. 8 Coverage achievements of GA-based test generation framework and RT on P_2

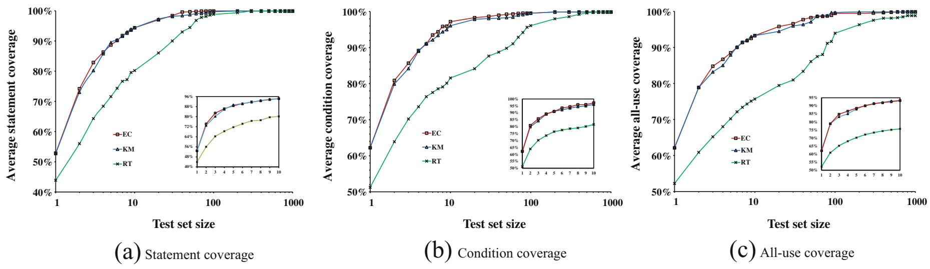


Fig. 9 Coverage achievements of GA-based test generation framework and RT on P_3

It is apparent in this figure that the sizes of test sets generated by RT are larger than those generated by GA-based approaches. In addition, EC performs better than KM for all scenarios.

6.3 Fault Localization Effectiveness

In this section, we present the fault localization effectiveness by using test sets generated by GA-based test generation framework and by RT with respect to the eight fault localization techniques described in Section 2.4. In Section 6.3.1, we focus on products with a single bug; in Section 6.3.2, multi-fault products are considered. X-RT, X-EC, and X-KM represent a specific fault localization technique discussed in Section 2.4 with the test sets generated by RT, EC, and KM, respectively.

6.3.1 Products With a Single bug

For most of the scenarios in Table 10, EC and KM outperform RT significantly. RT is more effective than either EC or KM in merely eight of the 96 scenarios. Note also that RT's enhancement in these eight scenarios is insignificant with the largest difference being less than 5%: with respect to P_2, Crosstab-RT requires the examination of 158.18 statements in the worst case, while Crosstab-KM requires 165.55.

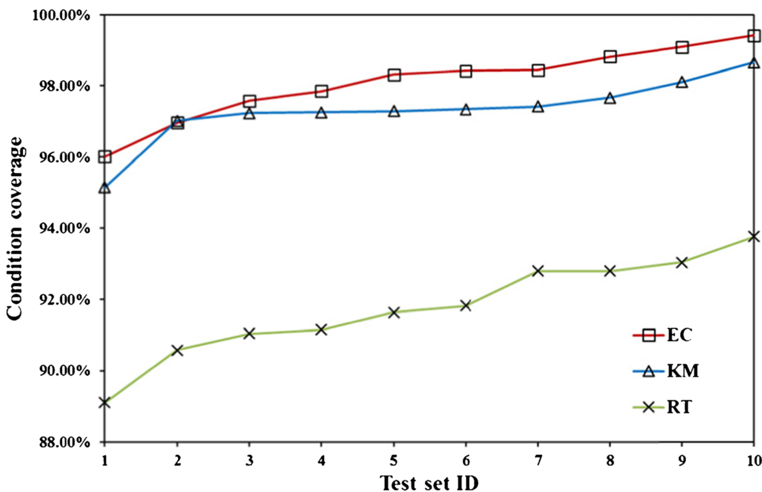


Fig. 10 Condition coverage of test sets with 60 test cases for P_3

Table 9 Average number of test cases that achieve specific coverage for J_1 to J_4

	Approach	Statement coverage			Condition coverage		
		85%	90%	95%	85%	90%	95%
J_1	EC	140.8	148.0	167.3	168.4	198.3	289.4
	KM	149.2	156.8	166.9	165.3	217.5	298.3
	RT	160.7	172.8	185.6	188.3	216.7	308.8
J_2	EC	125.7	135.2	157.1	135.1	167.2	250.1
	KM	132.6	145.8	155.2	150.6	170.5	265.8
	RT	167.4	178.6	190.3	197.4	245.8	310.3
J_3	EC	122.5	130.4	159.8	132.5	155.6	240.7
	KM	133.8	144.7	153.9	145.8	154.8	262.6
	RT	170.2	187.2	182.4	202.7	252.4	299.6
J_4	EC	127.5	135.3	155.7	135.5	152.3	288.1
	KM	124.8	139.7	159.4	137.8	157.7	289.9
	RT	168.8	180.5	187.3	189.6	248.1	345.8

Differences can also be observed between EC and KM. KM exceeds EC in 64.6% of all scenarios (62 of 96 scenarios). For example, with respect to P_5, Tarantula-KM exceeds Tarantula-EC in both the best case (65.67 vs 80.35) and the worst case (151.10 vs 172.03).

With respect to different fault localization techniques, results also vary:

- When using H3c and O^P, KM exceeds EC in more than 80% of the scenarios. For example, with respect to the 12 scenarios using O^P, KM shows better performance over EC in 10 of them, while EC exceeds KM only in the best case and worst case for P_5;
- When using Crosstab, EC exceeds KM in 8 out of the 12 scenarios.

Experiments are also performed to evaluate the proposed framework using EXAM score. For discussion purposes, the EXAM scores with respect to E_1 to E_2 in *EM* using H3b, Tarantula, and Crosstab are presented in Figs. 12 and 13. The horizontal axis represents the percentage of statements examined while the vertical axis represents the cumulative percentage of faulty versions where faults can be located by examining the corresponding percentage of statements marked on the horizontal axis. For example, with respect to Fig. 12a, by examining 10% of the code, H3b-EC can locate 83.33% of the faults in the faulty versions of E_1 in the best case and 76.67% in the worst case. In contrast, for H3b-RT, the best case is 79.89%, and the worst case is 70.89%.

Based on these figures, we observe the following:

- For E_1, the EXAM scores of EC outperforms KM and RT in most of the scenarios with only one exception in Fig. 12d, where Tarantula-EC, Tarantula-KM, and Tarantula-RT are comparable in the worst case;
- For E_2, EC and KM show significant improvements over *RT*. For example, in Fig. 13, the curves of H3b-KM and K3b-EC in the best case diverge significantly from the H3b-RT curve;
- For E_1 and E_2, the curves for RT show no obvious changes (because no test cases were reused); however, with respect to EC and KM, the effectiveness of reuse is significant when test cases for E_1 are reused for E_2.

Due to space limit, only the figures for E_1 and E_2 are presented. However, results for other programs and techniques (H3c, Ochiai, D^3, O, and O^P) also support the

Table 10 Average number of statements that need to be examined regarding P_1 to P_6

	Best case						Worst case					
	P_1	P_2	P_3	P_4	P_5	P_6	P_1	P_2	P_3	P_4	P_5	P_6
H3b-RT	84.58	90.41	91.91	134.86	111.33	89.86	150.91	150.78	163.85	211.79	199.81	159.33
H3b-EC	74.61	80.23	76.82	124.91	80.81	80.82	141.81	140.23	114.48	187.96	172.25	141.19
H3b-KM	75.97	79.05	80.13	108.24	90.57	75.60	140.79	135.66	107.61	176.65	186.48	134.38
H3c-RT	80.46	82.40	94.79	130.79	98.59	117.29	155.88	171.19	224.71	236.18	204.16	223.23
H3c-EC	72.49	61.23	80.33	103.19	82.19	90.13	135.86	107.58	111.18	183.33	167.53	167.18
H3c-KM	71.98	56.10	72.38	99.05	82.38	89.05	129.18	126.78	107.48	179.98	154.13	154.06
Crosstab-RT	94.69	101.40	96.40	110.40	139.93	90.43	168.46	158.18	171.96	200.03	224.91	179.48
Crosstab-EC	95.48	90.34	79.57	82.21	119.13	70.85	172.67	149.87	132.62	144.19	174.82	148.84
Crosstab-KM	89.72	95.29	84.05	90.53	123.69	67.33	167.97	165.55	147.14	164.88	185.84	138.38
Tarantula-RT	140.05	130.79	100.79	165.88	128.54	107.75	211.79	201.51	167.19	269.80	213.16	217.84
Tarantula-EC	110.19	109.57	91.05	150.33	108.35	95.91	200.41	179.59	137.38	270.19	172.03	178.53
Tarantula-KM	121.25	122.60	89.60	128.34	95.67	89.03	227.18	197.49	127.69	228.52	151.10	187.78
Ochiai-RT	90.10	83.45	93.84	117.45	100.45	97.62	174.48	143.04	164.21	213.91	193.88	211.33
Ochiai-EC	81.18	73.96	59.48	93.96	89.96	88.98	154.84	131.72	148.41	142.34	154.35	161.05
Ochiai-KM	79.33	56.62	60.16	80.48	81.19	95.67	144.33	108.51	153.87	137.72	148.82	182.34
D^3-RT	81.91	92.84	105.45	169.05	94.63	113.29	134.87	183.96	191.69	311.93	171.56	209.18
D^3-EC	79.67	76.80	90.38	121.63	90.03	96.23	129.67	164.67	173.67	274.29	149.62	167.05
D^3-KM	82.80	71.48	89.88	119.42	92.84	90.28	137.88	127.49	163.62	242.48	136.35	188.66
O-RT	80.59	82.06	140.81	132.87	87.91	117.26	157.67	163.88	231.29	235.69	149.60	267.79
O-EC	75.91	70.84	105.84	102.69	63.82	108.47	131.80	127.58	196.98	196.62	124.84	247.82
O-KM	74.92	75.57	98.32	97.57	72.05	90.87	126.43	120.47	208.16	203.05	151.97	202.05
O^P-RT	90.52	85.49	120.29	119.45	97.52	89.48	173.29	157.91	219.29	254.79	163.85	158.69
O^P-HM	75.63	76.36	95.87	101.92	88.56	80.78	149.92	131.03	168.51	205.48	124.48	139.21
O^P-KM	72.47	72.57	85.59	91.87	90.20	72.57	142.03	119.84	157.67	199.29	131.57	127.69

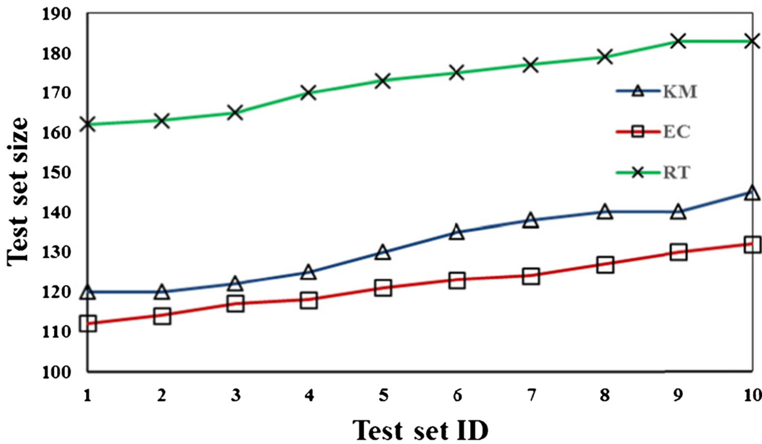


Fig. 11 Number of test cases that achieve 85% statement coverage for J_3

conclusion that EC and KM outperform RT in most of the scenarios. In addition, test sets generated by KM also show better performance in locating bugs than those by EC.

6.3.2 Products With Multiple Bugs

In the previous section, we apply our framework as well as RT to programs with exactly one fault. In this section, we move on to illustrate the effectiveness of our framework when dealing with programs with multiple bugs. Aside from the average number of statements that need to be examined to find the first bug, we also adopt two evaluation methods in this section, namely one-fault-at-a-time approach and *Expense* score-based approach.

The one-fault-at-a-time approach is an iterative method. In each iteration, one fault is located, and then the test set is re-executed to detect subsequent failures in order to locate and fix the next fault. The process continues until no failure is observed using the current test set. The *Expense* score-based approach concentrates on calculating the percentage of code that needs to be examined in order to locate the first bug. The authors of (Yu et al. 2008) argue that the first located bug is where programmers begin to fix and therefore the focus should be placed on locating the first bug. According to the description in Section 5.3, the expense score is equivalent to the EXAM score used in this paper.

Product versions with multiple faults are generated by combining several single-fault versions in various ways. For example, five distinct faults are taken from five randomly selected, single-fault versions and injected into the original product version to form a five-fault version. To avoid possible bias in creating products with multiple faults, 40 distinct faulty versions containing 4, 5, 6, and 7 faults, respectively, are created for H_1 to H_6. Altogether we have 960 product versions with multiple faults.

In order to make a more realistic experimental setup for SPLs, between two successive products, for example, H_1 to H_2, we randomly select one or several faults existing in H_1 to be injected into H_2. However, this does not indicate that at least one

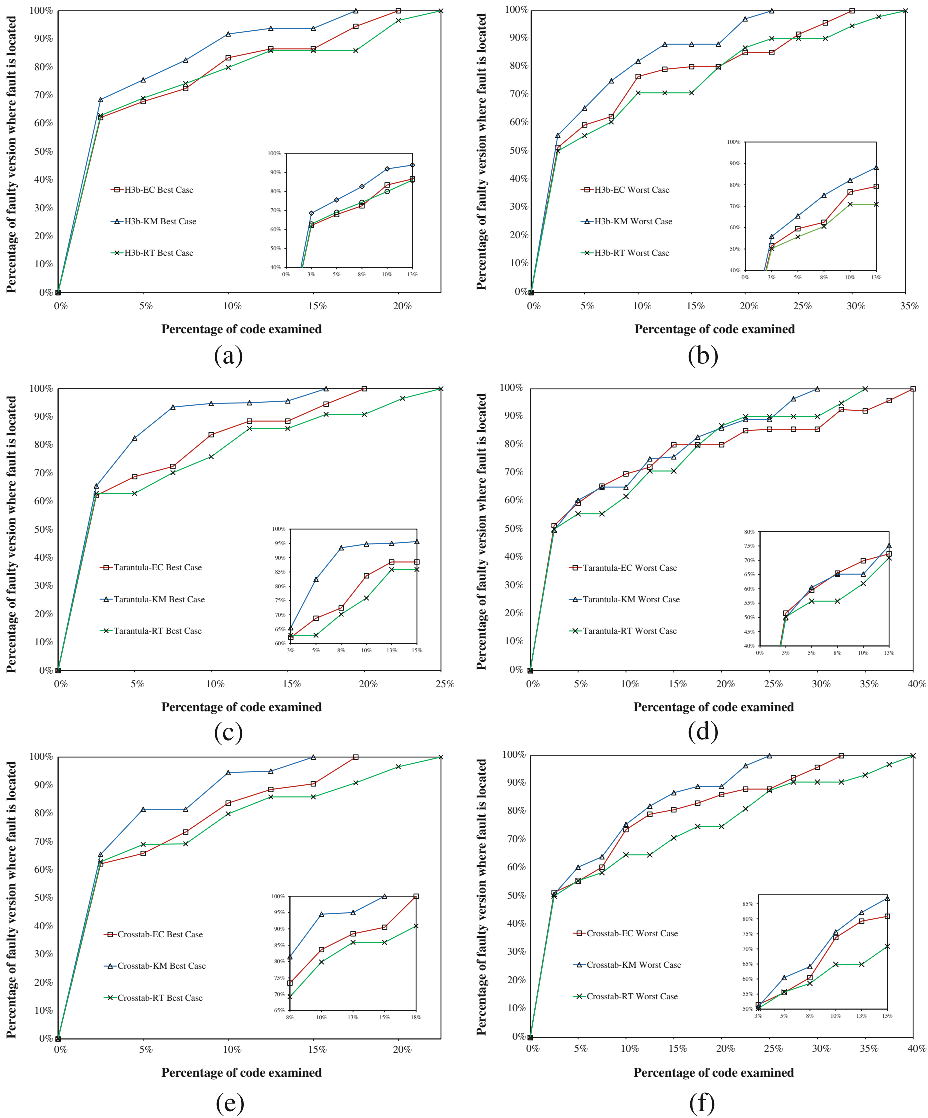


Fig. 12 EXAM score-based comparison among X, X-EC, and X-KM on E_1. (**a**) Best case using H3b, (**b**) Worst case using H3b, (**c**) Best case using Tarantula, (**d**), Worst case using Tarantula, (**e**) Best case using Crosstab, (**f**) Worst case using Crosstab

fault in H_1 will necessarily be injected into H_2. Of all the 800 faulty versions for H_2 to H_6, 15.50% (124 of 800 versions) do not share the same faults with its corresponding predecessor, while the rest have at least one fault from its corresponding predecessor injected.

Tables 11 and 12 represent the average number of statements that need to be examined to locate the first bug for the best and worst case, respectively. For example, in the best case, by using H3b-KM, an average of 78.50 statements need

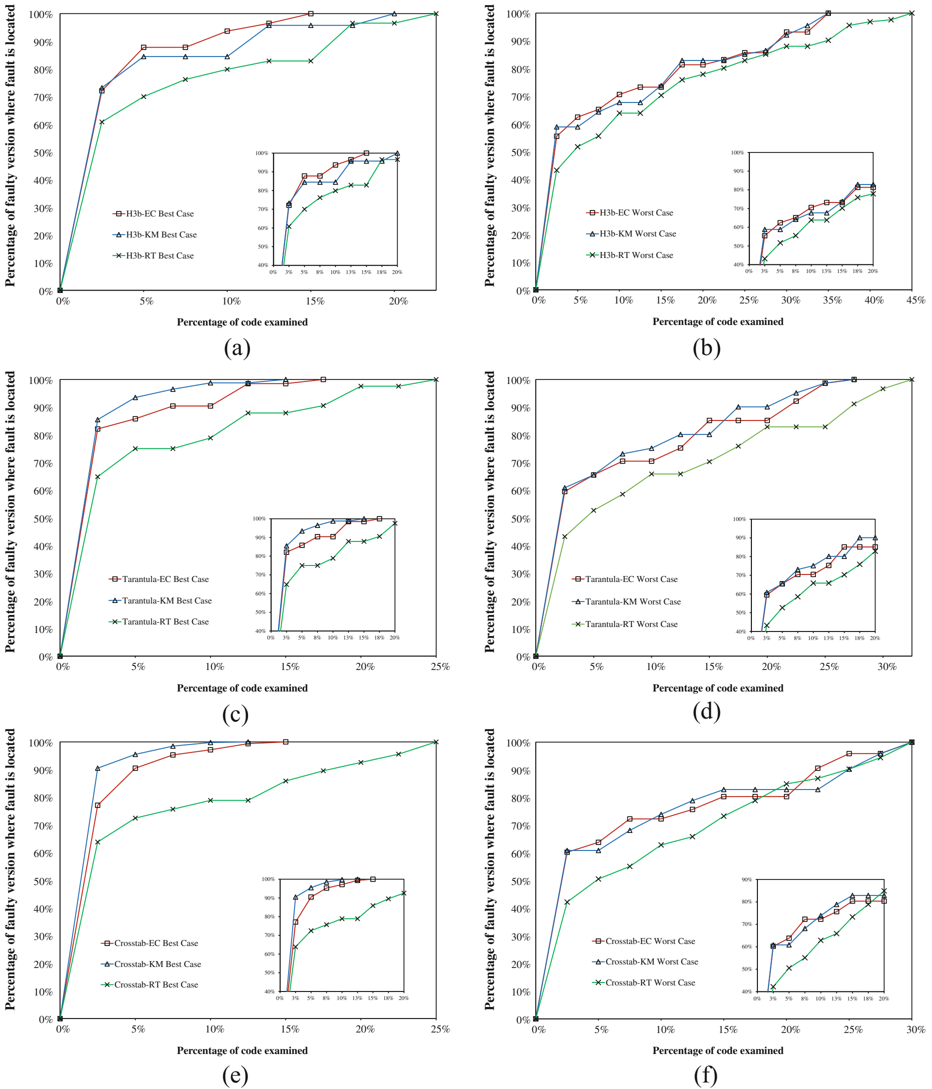


Fig. 13 EXAM score-based comparison among X, X-EC, and X-KM on E_2. (**a**) Best case using H3b, (**b**) Worst case using H3b, (**c**) Best case using Tarantula, (**d**), Worst case using Tarantula, (**e**) Best case using Crosstab, (**f**) Worst case using Crosstab

to be examined in order to locate the first bug for the 6-bug versions of H_2. On the other hand, the number is 149.03 in the worst case.

The following observations can be made based on the two tables:

- For the best case (Table 11), KM outperforms RT in 177 out of 192 scenarios (92.2%). EC is more effective than RT in 161 scenarios (83.9%). In addition, KM outperforms EC in 151 scenarios (78.6%).

Table 11 Average number of statements that need to be examined to locate the first bug (best case)

Best case	H_1				H_2				H_3				H_4
	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug
H3b-RT	75.38	70.28	80.53	60.68	70.43	73.88	85.90	75.63	32.86	26.18	25.45	40.20	44.28
H3b-EC	78.23	40.48	53.53	62.03	58.63	68.63	80.08	69.80	31.63	26.13	20.78	16.12	25.32
H3b-KM	69.68	35.45	45.28	53.43	68.58	65.83	78.50	58.70	23.23	18.48	9.60	6.88	11.80
H3c-RT	128.53	108.55	99.13	105.95	91.38	90.13	95.13	78.13	36.85	34.73	37.93	45.65	70.20
H3c-EC	90.85	75.50	105.23	101.00	80.23	75.85	90.40	93.30	29.35	26.58	33.33	19.28	59.73
H3c-KM	85.95	80.57	97.88	89.60	75.60	69.45	80.85	65.73	34.33	22.60	16.23	12.68	21.85
Crosstab-RT	79.80	82.75	98.83	95.98	75.20	108.65	106.20	108.45	20.63	33.70	45.00	33.40	66.85
Crosstab-EC	69.80	62.45	87.35	100.18	95.83	93.75	108.80	70.50	12.52	18.83	20.88	17.45	16.80
Crosstab-KM	65.85	70.08	76.27	99.35	89.80	85.13	75.03	76.80	13.32	12.63	8.15	6.25	11.58
Tarantula-RT	73.10	72.45	65.13	69.38	58.13	88.73	59.73	100.08	29.65	35.45	30.38	27.88	68.48
Tarantula-EC	53.55	50.15	60.80	68.50	48.78	56.53	28.53	35.98	26.85	19.93	28.63	30.75	65.13
Tarantula-KM	48.80	45.03	53.98	77.90	36.30	36.45	32.88	24.30	22.00	20.55	18.55	17.83	50.63
Ochiai-RT	55.65	73.78	48.53	102.80	79.68	40.85	79.63	80.83	30.85	33.58	29.63	25.73	65.83
Ochiai-EC	19.83	40.53	20.23	49.03	48.60	19.15	58.98	48.14	26.80	24.18	30.13	22.10	45.00
Ochiai-KM	11.25	25.85	34.15	58.55	24.98	28.36	45.55	34.10	14.58	12.05	11.83	10.25	40.53
D^3-RT	85.38	85.78	46.45	38.30	78.23	85.85	78.08	79.44	40.23	55.23	48.63	39.75	80.83
D^3-EC	109.82	76.72	56.23	59.03	42.65	39.28	40.93	55.28	26.48	25.13	23.13	19.20	65.00
D^3-KM	79.85	65.85	38.28	42.55	40.80	38.35	45.88	50.23	18.50	16.75	14.88	12.88	71.05
O-RT	58.08	105.58	95.73	65.48	79.80	73.55	70.35	101.78	40.30	50.75	49.58	45.98	55.68
O-EC	48.38	53.52	65.80	65.08	49.83	39.65	39.75	25.13	44.23	47.48	53.53	40.20	44.50
O-KM	58.60	75.28	85.93	49.03	25.80	24.35	21.56	21.54	38.33	36.43	35.95	26.88	36.48
O^P-RT	69.88	103.83	107.80	98.53	98.65	135.55	68.95	97.78	65.93	39.58	38.33	53.45	211.00
O^P-EC	68.46	73.25	79.03	73.85	55.80	80.83	58.28	51.53	68.28	45.95	43.40	38.65	135.33
O^P-KM	55.80	65.22	87.28	65.95	35.73	71.75	49.80	50.75	70.98	42.63	33.93	28.85	128.87

Best case	H_4			H_5				H_6			
	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug
H3b-RT	53.65	43.75	98.98	101.75	121.68	75.08	66.82	77.63	95.80	112.85	101.80

Table 11 (continued)

Best case	H_4			H_5				H_6			
	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug
H3b-EC	48.75	53.80	65.18	82.70	101.50	80.50	90.10	70.58	75.25	90.18	95.28
H3b-KM	35.88	45.85	53.75	98.60	68.28	78.35	58.65	65.80	70.28	81.95	75.65
H3c-RT	65.75	78.25	108.03	120.95	160.60	125.90	79.48	85.13	93.53	85.48	125.75
H3c-EC	48.40	70.25	50.98	95.28	108.58	85.23	95.18	64.85	73.48	101.85	108.65
H3c-KM	40.65	65.85	48.75	58.85	85.90	79.82	82.23	55.80	65.75	64.23	102.85
Crosstab-RT	59.35	58.53	27.58	40.70	85.50	75.50	63.10	98.93	70.58	108.33	97.45
Crosstab-EC	35.48	42.45	22.03	65.28	75.23	60.32	55.70	79.68	40.85	79.63	80.83
Crosstab-KM	23.28	32.98	19.03	35.80	68.05	58.75	35.28	79.80	55.13	85.03	86.80
Tarantula-RT	78.58	72.58	95.00	72.75	68.68	58.08	46.80	68.65	75.48	76.53	92.53
Tarantula-EC	65.05	35.83	25.25	55.80	44.93	37.95	18.15	54.80	48.80	46.93	89.68
Tarantula-KM	37.78	37.45	45.00	37.80	20.23	35.08	16.38	45.80	53.98	65.73	75.03
Ochiai-RT	68.60	34.85	45.85	108.05	81.58	108.58	98.63	124.63	138.30	115.08	98.43
Ochiai-EC	63.65	72.45	40.80	65.38	76.23	84.20	85.30	101.68	108.00	95.58	70.53
Ochiai-KM	58.38	65.65	23.20	45.28	53.23	65.58	95.03	85.80	95.60	83.63	75.83
D^3-RT	83.60	64.90	81.78	110.85	92.85	95.95	95.60	56.23	64.13	75.05	85.13
D^3-EC	56.38	32.43	68.55	86.38	86.23	94.20	80.30	40.43	39.20	42.43	56.13
D^3-KM	77.23	55.28	38.65	101.20	98.65	108.80	127.28	34.80	40.80	54.68	60.53
O-RT	55.85	53.93	54.50	102.40	116.63	118.65	82.95	13.25	28.80	40.83	70.03
O-EC	46.78	54.85	64.80	73.68	85.23	95.28	85.45	9.85	25.68	35.53	43.80
O-KM	39.40	51.00	50.00	65.80	70.85	85.98	70.65	11.85	13.25	27.55	33.48
O^P-RT	108.60	142.80	161.48	91.40	98.87	104.50	105.68	38.53	46.50	53.60	83.55
O^P-EC	158.43	135.90	118.42	80.20	73.93	86.60	68.98	40.80	53.73	71.73	73.30
O^P-KM	92.68	104.35	108.78	99.78	64.25	73.15	65.15	21.08	25.10	35.05	68.60

Table 12 Average number of statements that need to be examined to locate the first bug (worst case)

Worst case	H_1				H_2				H_3				H_4
	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug
H3b-RT	135.33	116.68	153.03	99.85	147.28	152.65	167.23	141.75	87.48	65.33	52.58	72.25	98.53
H3b-EC	112.85	95.23	118.85	136.25	121.25	138.33	158.73	138.75	62.38	50.98	47.70	60.53	78.28
H3b-KM	108.83	72.80	80.33	98.63	135.80	129.35	149.03	127.65	53.73	40.28	42.43	32.05	38.70
H3c-RT	209.68	189.70	175.35	197.13	174.60	185.23	191.93	169.73	80.98	73.45	56.25	90.13	118.53
H3c-EC	172.35	191.23	180.33	187.15	166.33	153.73	178.45	157.85	75.53	69.23	58.48	85.75	107.18
H3c-KM	159.85	179.75	169.28	176.43	158.23	147.28	171.38	138.18	83.23	43.85	43.85	38.78	65.58
Crosstab-RT	145.75	159.58	168.73	179.93	137.03	198.68	201.53	215.83	55.78	63.53	78.50	83.35	115.43
Crosstab-EC	129.85	137.60	157.20	202.85	168.35	178.73	188.75	173.50	45.38	50.73	43.23	58.73	40.88
Crosstab-KM	115.23	124.18	143.38	181.63	169.80	153.30	170.98	157.80	50.65	40.25	38.75	46.35	37.05
Tarantula-RT	120.15	106.25	156.65	115.13	117.28	165.65	134.45	193.75	60.85	70.48	65.85	58.18	137.13
Tarantula-EC	109.93	103.80	123.73	107.15	95.75	127.55	85.53	82.90	53.75	45.23	56.93	63.73	124.75
Tarantula-KM	98.90	90.48	99.38	132.05	70.30	95.45	70.83	65.30	46.13	43.83	47.08	48.65	50.63
Ochiai-RT	97.58	124.25	92.63	215.75	127.85	93.73	157.63	148.83	78.65	80.63	63.18	59.23	127.28
Ochiai-EC	50.83	91.28	45.03	107.18	90.83	60.28	102.65	85.20	62.13	59.35	73.28	65.28	101.70
Ochiai-KM	37.58	53.78	60.73	127.20	49.85	55.43	92.38	71.23	63.23	65.65	58.18	45.75	95.80
D^3-RT	128.85	139.83	90.23	82.83	157.85	147.58	142.18	161.35	79.23	107.75	105.25	85.65	160.75
D^3-EC	139.78	138.70	98.45	101.05	99.98	91.83	87.43	107.63	52.83	78.28	75.35	78.20	115.15
D^3-KM	99.43	114.03	75.78	79.23	87.70	80.43	76.00	90.05	49.28	64.85	52.35	52.35	135.13
O-RT	687.75	785.65	557.80	443.75	793.79	723.45	458.67	558.75	897.58	798.95	658.75	597.28	757.15
O-EC	640.57	573.13	490.78	349.85	803.83	597.75	394.83	754.35	754.68	572.85	792.83	825.98	654.28
O-KM	853.60	497.58	459.13	409.23	571.23	327.80	212.30	357.53	754.90	820.54	352.00	279.28	638.57
O^P-RT	138.55	212.83	208.25	193.35	159.43	211.78	118.83	173.48	187.50	98.73	101.53	109.58	325.28
O^P-EC	112.65	175.53	179.30	153.85	101.38	157.25	112.73	108.65	138.35	108.20	95.28	85.75	246.85
O^P-KM	105.80	112.23	172.65	136.98	68.80	142.43	93.35	96.05	145.85	98.73	68.73	68.05	202.18

Worst case	H_4			H_5				H_6			
	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug
H3b-RT	118.83	93.50	185.23	185.28	254.75	154.80	105.83	138.58	170.83	187.53	195.80

Table 12 (continued)

Worst case	H_4			H_5				H_6			
	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug	4-bug	5-bug	6-bug	7-bug
H3b-EC	92.93	107.48	132.83	167.75	201.45	138.15	170.55	127.73	157.53	175.80	185.85
H3b-KM	73.57	85.10	95.80	172.53	131.73	124.95	101.98	118.20	145.80	165.53	175.03
H3c-RT	132.53	138.25	195.53	197.85	271.80	198.75	129.93	165.35	175.38	153.84	195.73
H3c-EC	98.33	158.28	125.38	187.80	248.00	178.33	148.73	137.50	154.85	187.58	211.03
H3c-KM	87.35	124.53	98.63	112.48	168.73	157.63	158.78	118.03	134.53	98.08	232.40
Crosstab-RT	108.23	137.95	80.70	75.50	160.25	148.57	127.70	175.38	112.35	185.60	175.10
Crosstab-EC	75.85	96.75	53.65	95.08	154.13	135.28	114.20	157.05	95.48	154.70	158.28
Crosstab-KM	50.65	70.75	49.23	68.75	128.23	118.73	89.35	145.28	102.75	148.18	149.68
Tarantula-RT	140.25	158.85	170.73	138.40	140.75	124.28	101.75	108.73	134.28	148.35	176.03
Tarantula-EC	129.28	83.53	68.25	102.75	108.70	89.15	53.28	95.40	102.28	105.73	167.43
Tarantula-KM	37.78	78.45	89.00	108.75	99.23	85.28	45.18	85.03	115.45	124.93	143.28
Ochiai-RT	163.33	89.75	95.43	198.53	138.50	185.95	185.63	211.43	221.58	195.88	175.93
Ochiai-EC	137.33	127.53	110.76	132.85	140.53	175.95	170.23	185.85	197.40	185.50	165.45
Ochiai-KM	142.23	110.98	58.28	128.23	117.13	148.80	165.63	175.98	185.35	163.25	153.85
D^3-RT	154.73	115.25	156.85	209.48	195.75	175.85	175.53	98.15	110.33	125.23	138.65
D^3-EC	99.73	85.30	129.95	143.85	148.53	178.00	168.35	101.80	80.85	117.83	108.75
D^3-KM	121.63	101.23	76.85	158.63	153.63	185.75	198.85	78.03	79.43	98.78	128.38
O-RT	678.73	715.95	574.23	854.67	434.75	397.79	795.28	843.75	689.79	485.45	458.67
O-EC	435.55	398.70	308.95	629.83	492.88	726.83	885.93	397.58	459.13	409.23	571.23
O-KM	537.13	485.78	493.85	212.30	309.23	275.20	515.28	485.65	557.80	453.78	793.79
O^P-RT	211.38	279.68	295.53	173.85	180.87	211.23	187.73	85.45	90.93	105.73	158.03
O^P-EC	307.05	254.08	207.20	163.35	158.25	178.53	140.85	89.35	95.25	112.78	148.20
O^P-KM	198.15	178.65	207.85	158.65	129.25	153.15	138.28	70.23	75.80	85.53	115.25

- For the worst case (Table 12), KM outperforms RT in 179 out of 192 scenarios (93.2%). EC is more effective than RT in 155 scenarios (80.7%).
- Differences can also be observed between KM and EC. In the best case (Table 11), KM shows better performance than EC in 151 scenarios (78.6%); in the worst case (Table 12), KM defeats EC in 153 scenarios (79.7%).

Next, Figs. 14 and 15 present the comparison among EC, KM, and RT based on EXAM score, which can also be referred to as *Expense* score when dealing with products with multiple faults. For example, referring to Fig. 15, for H_4, by examining 2% of the code, H3b-EC can locate the first bug in 75.58% of versions with multiple faults in the best case; in contrast, the number is for H3b-RT we can only locate the first bug in 68.80% of multiple-fault versions.

Due to space limit, we present here only the figures for H_3 and H_4 using H3b, Tarantula, and Crosstab. However, we examine figures for other products using all eight fault localization techniques and make the following observations:

- For most of the scenarios, KM shows better performance than RT in locating the first bug within a program with multiple bugs;
- For most of the scenarios, EC also outperforms RT;
- KM also shows an advantage over EC in most of the scenarios.

Finally, we examine the effectiveness of our proposed framework using the one-fault-at-a-time approach. A multi-fault version with five faults is randomly selected from the 5-bug versions of J_4. To avoid biased conclusion, 40 test sets of size 100 are then generated for this version using EC, KM, and RT, respectively. Table 13 presents the average number of statements that need to be examined in each iteration to locate all the faults sequentially.

From Table 13, we observe that in most of the scenarios, the cumulative total of average number of statements that need to be examined by KM is less than EC and RT. Only two exceptions are found: in the best case, Crosstab-KM requires 220.90 statements examined to locate all five bugs while Crosstab-EC needs 219.01; in the worst case, the number for H3b-KM is 251.36 while 249.07 for H3b-EC.

In addition, with respect to each iteration, KM also outperforms EC and RT, which indicates that test sets generated using KM are more effective in locating bugs for multi-fault programs when compared to EC and RT.

7 Performance of EC and KM

In this section, we present the time needed to generate a number of test cases by using EC and KM. Without loss of generality, we employ J_2, which consists of 64,857 lines of code, to demonstrate the efficiency of our proposed framework. For a given test set size (as shown by the *x*-axis of Fig. 16), 30 distinct test sets are generated and the average time to generate these test cases are collected. The experiments are performed on a desktop with Intel Core i7-3770 and 16G Memory, running Ubuntu 16.04.

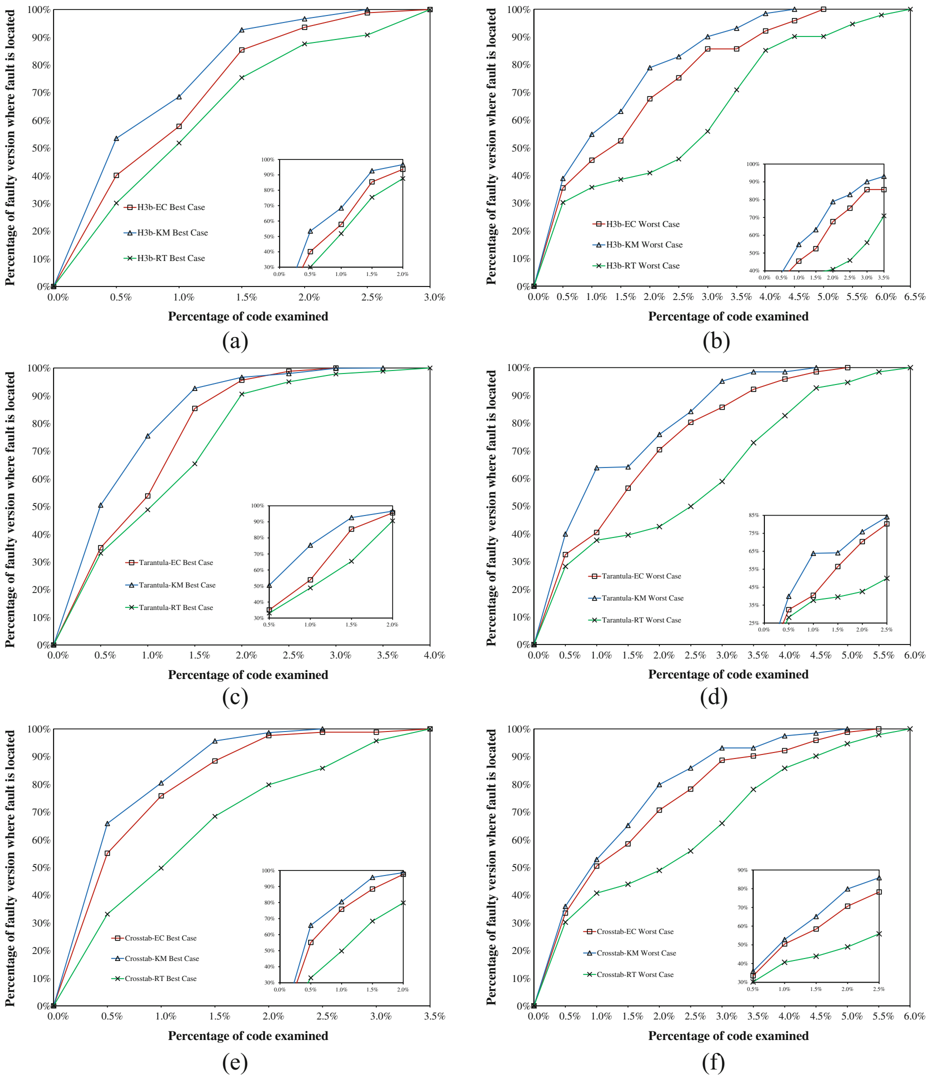


Fig. 14 EXAM score-based comparison among X, X-EC, and X-KM on H_3. (**a**) Best case using H3b, (**b**) Worst case using H3b, (**c**) Best case using Tarantula, (**d**), Worst case using Tarantula, (**e**) Best case using Crosstab, (**f**) Worst case using Crosstab

According to Fig. 16, little time is needed to generate these test sets when their sizes are samll (less than 100) by using EC and KM. For example, with respect to the KM approach, an average of 4.617 s are used to generate a set of 50 test cases. In contrast, EC only needs 2.358 s.

With the increase of test set size, the average time to generate these sets also increases. For example, with respect to the KM approach, the time increases from 43.597 s for a set of 200 test cases to 1,434.842 s for a set of 1,000 test cases. Nevertheless, since the entire framework can be automated (in fact it has already been done in our studies) and the potential saving of

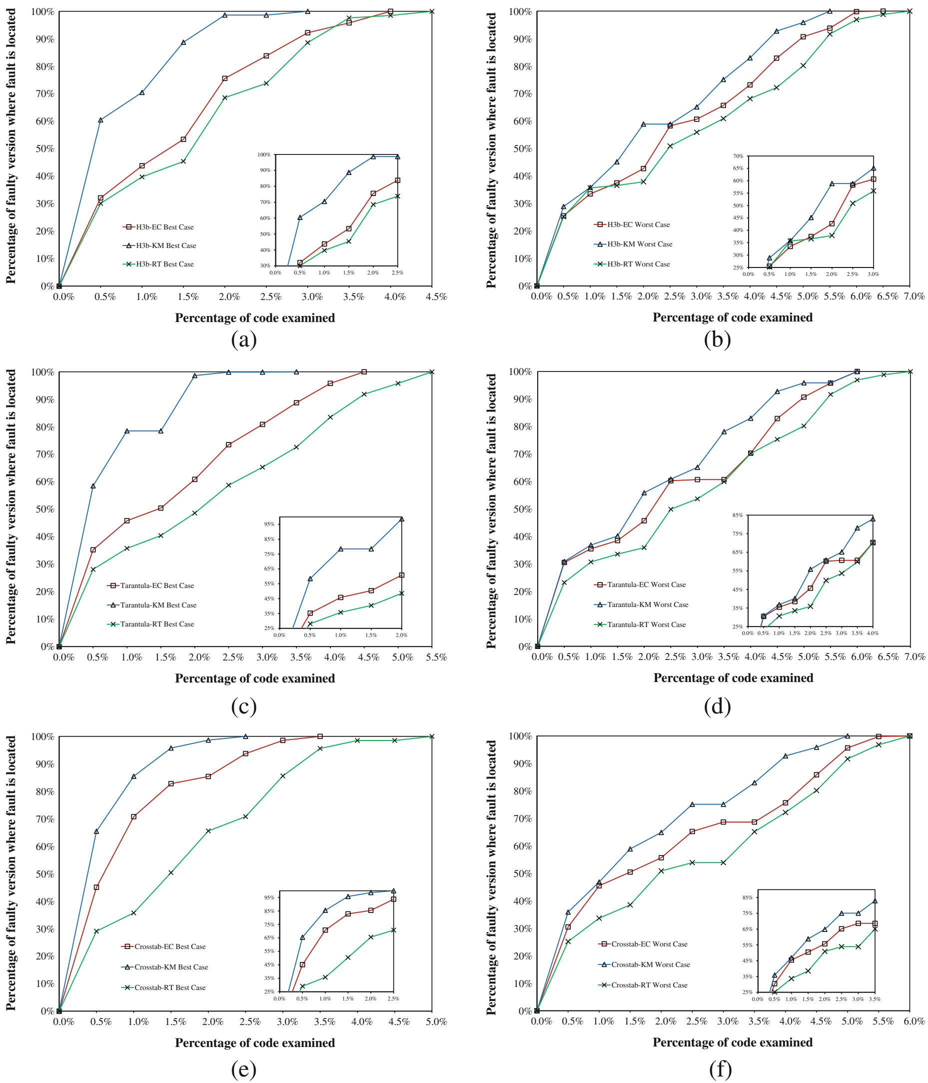


Fig. 15 EXAM score-based comparison among X, X-EC, and X-KM on H_4. (**a**) Best case using H3b, (**b**) Worst case using H3b, (**c**) Best case using Tarantula, (**d**), Worst case using Tarantula, (**e**) Best case using Crosstab, (**f**) Worst case using Crosstab

time and human effort in debugging by using test cases so generated, the proposed framework is applicable to real-life SPLs.

8 Discussion

In this section, we discuss some interesting factors that may affect the effectiveness of the proposed framework.

Table 13 Number of statements that need to be examined following the one-fault-at-a-time approach

		Average number of statements examined in each iteration					Cumulative total			Average number of statements examined in each iteration					Cumulative total
		1	2	3	4	5				1	2	3	4	5	
H3b-RT	Best	28.23	33.65	45.78	69.45	47.65	224.76	Ochiai-RT	Best	39.58	48.50	58.80	90.93	58.95	296.76
	Worst	38.45	56.93	68.23	74.83	58.15	296.59		Worst	60.73	60.18	70.68	104.85	75.83	372.27
H3b-EC	Best	27.58	28.10	35.43	56.43	50.35	197.89	Ochiai-EC	Best	37.78	39.75	43.93	85.90	60.63	267.99
	Worst	34.43	39.75	46.88	70.13	57.88	249.07		Worst	55.58	52.00	65.83	95.80	75.30	344.51
H3b-KM	Best	25.43	27.50	33.85	49.85	47.80	184.43	Ochiai-KM	Best	35.48	35.65	55.98	80.83	49.45	257.39
	Worst	36.75	39.73	49.75	69.35	55.78	251.36		Worst	48.98	46.80	69.83	92.03	59.85	317.49
H3c-RT	Best	48.13	60.08	58.23	75.48	59.13	301.05	D^3-RT	Best	29.38	45.85	65.88	91.85	69.03	301.99
	Worst	60.58	75.58	80.38	90.63	100.45	407.62		Worst	46.65	58.90	75.80	103.18	80.86	365.39
H3c-EC	Best	36.35	49.88	60.53	68.70	48.68	264.14	D^3-EC	Best	24.93	39.83	55.83	70.58	59.80	250.97
	Worst	45.08	74.25	75.68	83.85	80.20	359.06		Worst	49.05	49.80	66.68	90.80	70.73	327.06
H3c-KM	Best	27.70	37.83	58.58	71.08	50.73	245.92	D^3-KM	Best	23.23	31.78	50.35	68.65	55.53	229.54
	Worst	48.28	48.13	68.73	85.48	85.25	335.87		Worst	35.73	43.52	65.80	89.68	69.85	304.58
Crosstab-RT	Best	33.65	43.28	60.13	59.60	56.73	253.39	O-RT	Best	18.48	26.70	38.45	65.15	45.78	194.56
	Worst	58.75	73.83	74.83	73.70	75.80	356.91		Worst	192.53	385.75	458.85	585.23	286.98	1909.34
Crosstab-EC	Best	26.25	36.25	51.53	55.63	49.35	219.01	O-EC	Best	19.80	25.70	38.05	63.70	40.85	188.1
	Worst	37.78	58.38	67.73	71.87	67.80	303.56		Worst	200.73	335.70	454.80	485.83	222.13	1699.19
Crosstab-KM	Best	24.58	39.63	48.33	62.83	45.53	220.90	O-KM	Best	15.53	23.23	43.83	60.85	39.73	183.17
	Worst	35.03	49.75	58.68	70.35	70.65	284.46		Worst	173.83	332.53	398.65	405.80	221.78	1532.59
Tarantula-RT	Best	32.68	48.65	65.90	90.58	56.28	294.09	O^P-RT	Best	20.85	25.23	39.65	75.93	58.08	219.74
	Worst	53.30	70.45	80.03	110.30	77.58	391.66		Worst	53.08	65.68	83.13	105.95	90.98	398.82
Tarantula-EC	Best	21.48	39.70	53.23	79.38	62.75	256.54	O^P-EC	Best	18.03	27.18	36.75	69.65	45.83	197.44
	Worst	43.65	51.85	69.75	95.65	91.68	352.58		Worst	40.23	38.85	49.93	85.13	85.03	299.17
Tarantula-KM	Best	20.68	38.45	50.68	81.23	59.85	250.89	O^P-KM	Best	17.13	25.20	34.00	63.70	43.08	183.11
	Worst	38.65	47.65	75.80	93.65	70.83	326.58		Worst	39.88	40.58	45.33	79.00	78.55	283.34

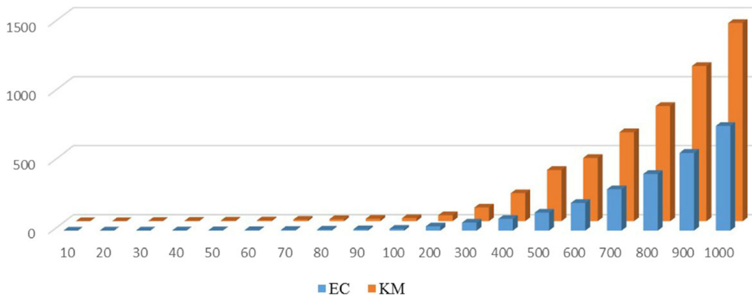


Fig. 16 Average time to generate 30 distinct test sets of a given size (the corresponding *x*-value) using EC and KM

8.1 Impact of Similarity Threshold on KM Approach

In our case study in Section 6, we set $\alpha = 0.1$ to compute the similarity threshold θ when using KM approach. Obviously, with various value of α, the effectiveness of the KM approach may vary. For instance, with the increase of α, the K-Means clustering results may be changed as the number of clusters varies. Consequently, the coverage achievement and fault localization effectiveness of test sets generated by KM could be affected.

As introduced in Section 3.2.2, the value of α has an impact on the determination for the number of clusters in calculating the fitness values of each test case. A larger α would decrease the number of clusters in each iteration of generating new test cases and increase the number of coverage vectors within a cluster. Consequently, the Euclidean distance between each coverage vector and its corresponding centroid increases. Since the above distance is negatively related to the fitness value of test cases, the fitness value assigned to each test case would decrease, especially for those with relatively high fitness values. As a result, the possibility of selecting test cases that are likely to create “good” test cases is lowered, which may weaken the effectiveness of KM approach.

To study the possible impacts, experiments were conducted on the 160 multi-fault versions of H_3 discussed in Section 6.3.2. The values of α are 0.10, 0.15, 0.20, 0.25, and 0.30.

To eliminate the possible impacts due to test case reuse, in this subsection, instead of following the procedure described in Section 3, we exclude the test case reuse process (test cases for H_2 are not reused for H_3).

First, we examine the possible impacts of α on coverage achievements. With respect to each version, multiple test sets are generated with size of 100 based on the three steps described in Section 5.2. Table 14 presents the average coverage achievements with respect to three coverage criteria (statement, condition, and all-use coverage). We observe that different values of α show no significant impact on coverage achievements.

Next, we examine the possible impacts of α on the fault localization effectiveness of KM. Tables 15 and 16 gives the average number of statements that needs to be examined in order to locate the first bug. We observe that the effectiveness of KM downgrades with α increases. For example, with respect to 5-bug versions, the average number of statements that need to be examined in the best case by H3c-KM increases with α increases. However, some exceptions also exist. For example, with respect to 5-bug versions, when increases from 0.15 to 0.20, the average number of statements that need to be examined in the best case by Ochiai-KM decreases from 13.25 to 9.85.

Though some exceptions exist, generally speaking, we observe that the effectiveness of KM reduces with α increases. Further study will concentrate on figuring out the optimal value of α.

Table 14 Coverage achievements on H_3 using different α

	α				
	0.10	0.15	0.20	0.25	0.30
Statement coverage	95.89%	96.21%	96.17%	95.19%	95.01%
Condition coverage	94.02%	93.67%	94.27%	93.98%	94.87%
All-use coverage	93.98%	95.09%	94.57%	94.89%	94.34%

8.2 Impact of Parameters β and γ

In Section 6, we assign 0.5 to both β and γ. In this subsection, we conduct experiments on the impacts due to different values of β and γ. First, we generate 30 test sets with a size of 200 for a randomly selected 5-bug version of H_2 using EC and KM, respectively. Then during the test case reuse process, different values of β and γ are assigned ranging from 0 to 1. The values of β are 0, 0.05, 0.10…0.90, 0.95, and 1.00; the corresponding values of γ are 1.00, 0.95, 0.90…0.10, 0.05, and 0. 10% of the test cases within each test set are reused to test H_3 based on Eq. (14) with different values of β and γ. The average statement coverage achievements as well as the effectiveness in locating bugs of these test cases are evaluated to explore the impacts of β and γ.

Figure 17 presents the average statement coverage achieved by the reused test cases using different values of β and γ. From the figure, we observe that the average statement coverage increases with β increases. For example, with $\beta = 0.2$, by using EC approach, the average statement coverage is 80.08%; with $\beta = 0.9$, the value increases to 86.88%. In addition, the coverage achievements increases significantly with β increases from 0 to 0.5; however, the coverage stabilizes with β larger than 0.5.

Note also that the impact of β and γ is more significant for EC approach. For KM approach, though similar trend can be observed, the differences are slight.

Figure 18 shows the average number of statements that need to be examined to locate the first bug using the 30 groups of 20 reused test cases. For discussion purposes, only figures for Tarantula-EC and Tarantula-KM are provided, though we also collected data using other metrics to make the following observations:

- The impact of β and γ on the effectiveness in locating bugs is insignificant for KM approach;
- For EC, with the increase of β, the average statements that need to be examined to locate the first bug also increases. For example, in Fig. 18a, with $\beta = 0.20$, the average number of statements that need to be examined by Tarantula-EC is 30.07 in the best case and 57.60 in the worst case; however, with $\beta = 0.90$, the number is 36.87 in the best case and 66.40 in the worst case. In other words, the effectiveness in locating bugs slightly reduces.

9 Related Studies

Software product line testing is a relatively new, but intense field of research since product line engineering has shown significant benefits in both academia and industry (Wang et al. 2012).

In (McGregor 2001), McGregor presented a set of activities which focus on testing the SPL assets (e.g. inconsistency between the requirements and implementation) and testing other

Table 15 Average number of statements need to be examined by X-KM using different α on multi-fault versions of H_3 (best case)

Best case	4-bug					5-bug					6-bug					7-bug				
	0.10	0.15	0.20	0.25	0.30	0.10	0.15	0.20	0.25	0.30	0.10	0.15	0.20	0.25	0.30	0.10	0.15	0.20	0.25	0.30
H3b-KM	23.23	26.65	32.38	45.63	36.43	18.48	25.78	25.63	32.48	38.55	9.60	12.70	16.03	15.51	18.68	6.88	9.63	10.53	10.43	8.38
H3c-KM	34.33	36.83	38.28	32.13	36.80	22.60	27.28	30.33	33.05	34.88	16.23	18.68	17.53	20.40	22.28	12.68	15.65	12.70	16.58	17.68
Crosstab-KM	13.32	15.83	8.65	13.88	9.58	12.63	13.05	8.45	10.00	15.42	8.15	10.18	12.68	13.85	11.08	6.25	5.85	8.65	9.48	11.08
Tarantula-KM	22.00	17.38	25.65	20.45	22.03	20.55	22.38	25.08	23.50	25.48	18.55	16.80	20.48	23.56	23.89	17.83	15.80	19.93	20.38	19.60
Ochiai-KM	14.58	16.63	10.83	16.85	12.45	12.05	13.25	9.85	12.95	13.35	11.83	12.85	13.28	12.38	14.85	10.25	11.65	12.25	10.50	10.80
DStar-KM	18.50	20.20	22.85	24.35	26.08	16.75	15.40	18.55	23.45	25.38	14.88	15.28	16.65	18.35	15.50	12.88	12.73	18.20	16.10	19.58
O-KM	38.33	40.38	39.20	42.80	45.45	36.43	38.48	40.45	42.35	39.85	35.95	36.68	40.58	43.50	49.88	26.88	30.28	32.10	25.48	30.50
O^P-KM	70.98	72.65	70.95	80.30	81.63	40.63	41.38	40.23	43.88	45.10	38.93	46.10	41.53	43.80	45.63	40.85	39.28	45.63	40.92	42.35

Table 16 Average number of statements need to be examined by X-KM using different α on multi-fault versions of H_3 (worst case)

Worst case	4-bug					5-bug					6-bug					7-bug				
	0.10	0.15	0.20	0.25	0.30	0.10	0.15	0.20	0.25	0.30	0.10	0.15	0.20	0.25	0.30	0.10	0.15	0.20	0.25	0.30
H3b-KM	41.28	44.75	58.30	65.80	54.65	36.85	40.68	46.55	58.62	65.30	33.48	34.33	32.84	39.72	40.45	23.78	24.42	26.62	27.64	22.53
H3c-KM	67.68	68.65	70.20	75.28	72.53	52.55	53.85	45.53	54.65	68.98	42.35	43.53	42.65	45.88	48.90	35.81	40.88	39.28	41.33	42.10
Crosstab-KM	39.15	40.28	41.03	42.58	43.38	33.93	36.29	35.23	35.05	38.03	23.85	25.68	23.33	33.45	28.03	28.65	29.58	30.99	32.73	30.65
Tarantula-KM	55.33	54.38	60.08	56.95	58.50	50.18	52.08	55.35	50.35	65.83	49.53	50.85	52.95	55.43	59.08	45.28	46.00	44.68	48.58	50.18
Ochiai-KM	38.25	39.50	40.85	38.43	39.58	34.83	38.65	33.88	40.75	41.88	30.65	29.78	35.98	32.85	39.88	29.58	30.85	32.88	30.45	33.45
DStar-KM	37.58	35.35	38.60	40.25	43.70	35.80	35.03	38.78	42.05	40.30	34.55	35.73	36.10	30.48	33.50	30.58	30.53	33.43	35.08	31.18
O-KM	78.80	80.40	90.08	85.03	83.93	70.85	75.15	82.55	83.93	90.60	67.65	70.85	70.05	72.88	68.95	65.58	63.85	60.48	74.25	75.93
O^P-KM	110.85	120.63	115.95	125.13	172.63	81.35	79.38	82.68	90.43	100.10	70.50	72.83	75.88	85.35	105.60	68.73	77.28	90.43	80.25	90.03

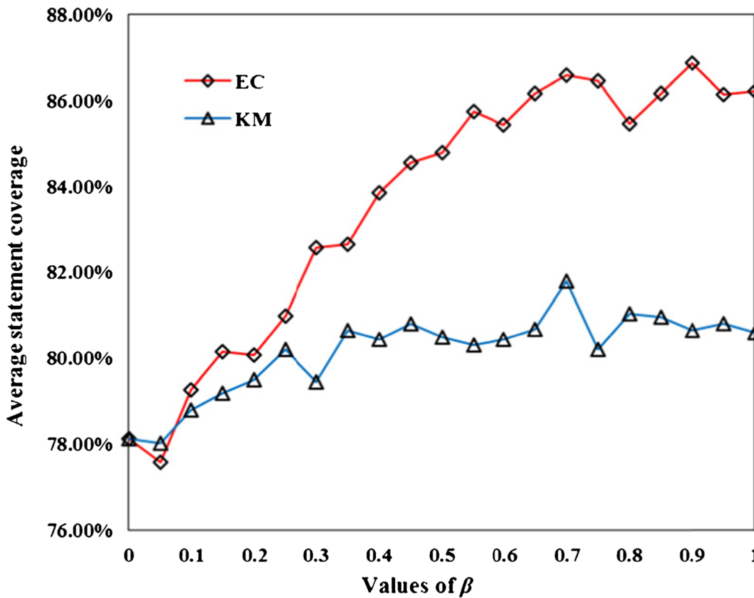


Fig. 17 Statement coverage on H_3 using different β and γ

artifacts (e.g. test-case derivation and test suite design), which represent complete products in the context of product line. In (Pohl and Metzger 2006), Pohl and Metzger explored six principles for SPL system testing to deal with the challenges when developing test strategies for SPL engineering. These principles discussed not only the obstacles in testing SPL but also provided possible solutions for engineers to thoroughly perform different techniques to assure the quality of an SPL.

Model-based test generation techniques using functional (Mohamed Ali and Moawad 2010; Nebut et al. 2003) and non-functional (Reis et al. 2006; Sinha et al. 2011) requirements for SPL have been proposed. The study by Nebut et al. (2003) focused on automated generation of functional test cases using requirements expressed in UML. Mohamed Ali and Moawad

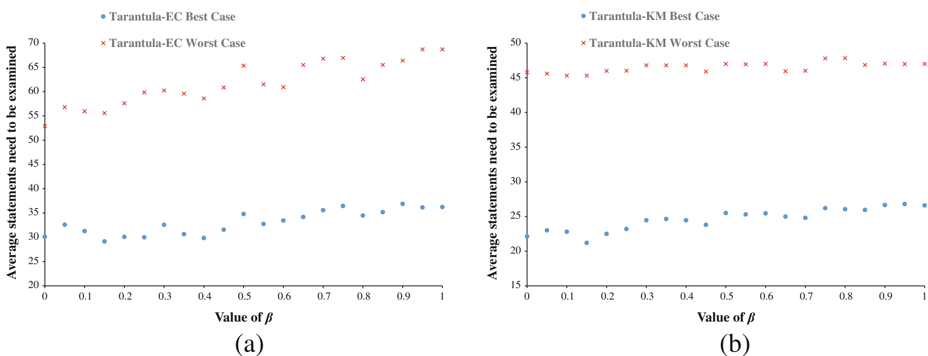


Fig. 18 Average number of statements that need to be examined by Tarantula-EC and Tarantula-KM using different β and γ. (**a**) Tarantula-EC. (**b**) Tarantula-KM

(2010) proposed an improved test generation based on two model-based techniques, PLUTO (Bertolino and Gnesi 2003) and V-Activity Diagram. In (Reis et al. 2006), Reis et al. described a technique that supports the development of reusable performance test case scenarios in domain engineering and the reuse of these scenarios in application engineering. In another paper (Lopez-Herrejon et al. 2015), Lopez-Herrejon et al. reported a systematic mapping study on applying combinatorial interaction testing to SPLs. In a more recent publication (Beohar et al. 2016), Beohar et al. discussed basic behavioral models for SPLs and proposed notions of SPL testing in order to precisely capture product derivations. In (Fischer et al. 2016), Fischer et al. conducted an empirical assessment of combinatorial interaction testing for SPLs. Mutation testing has also been applied to the field of model-based SPL testing. The idea of (Henard et al. 2014b) is mainly on generating test cases in order to find simulated faults in feature models.

Search-based test generation techniques have also been proposed to the area of SPL testing. In Harman et al. (2014), Harman et al. surveyed recent research activities for search-based software engineering for SPL and pointed out the future directions in this research area. A choice to testing a software product line is to use combinatorial interaction testing, first introduced by McGregor in (McGregor 2001). Later, other publications including (Henard et al. 2014a, 2013; Perrouin et al. 2012, 2010) introduced techniques to generate combinatorial tests for SPLs, as well as mechanisms to prioritize these test cases and tool support. Genetic algorithms have also been applied to the area of search-based SPL testing (Ensan et al. 2012; Guo et al. 2011). Authors of (Ensan et al. 2012) proposed an approach based on genetic algorithms to explore the configuration space of a software product line feature model, while work by Guo et al. in (2011) investigated a possible solution to optimize the feature selection in SPLs with resource constraints. Authors of (Wang et al. 2015) proposed a test suite minimization approach to reduce the number of test cases needed to test SPLs. In (Xu et al. 2013), Xu et al. proposed a test suite augmentation technique to generate new test cases for SPLs based on the uncovered branches in newly developed products.

Even though several approaches have been proposed with respect to testing of SPL, our approach is significantly different from related work. First of all, to the best of our knowledge, no existing publications integrate fault localization techniques into the test generation and reuse process to help better debug products of an SPL. Also, given the fact that most test generation approaches for SPL are based on a feature model, they suffer from limited practical value. Our approach, on the other hand, focuses on the effective reuse of test cases generated for one product in the newly developed product as well as the measurement of quality of the test cases, which benefits the process of developing a family of software products and further helps reduce the time and cost involved in testing and debugging of an SPL.

10 Threats to Validity

With respect to the proposed framework, we concede that the two approaches, EC and KM, are based on the following two assumptions, respectively:

- By using EC to generate test cases, we assume that the execution traces of test cases with larger Euclidean distances to other paths tend to be “far away” from those with smaller distances. The region around these paths is less likely to be covered by the current test cases. Therefore, if we select these test cases to generate new ones, we increase the chances

of covering the unexecuted regions. However, our approach does not guarantee we can always get the "optimal" test cases in each iteration which increase the coverage to the largest extent. In our case studies, the EC approach works very effectively in achieving high coverage with less test cases than RT.

- By using KM to generate test cases, the assumption is that if the execution trace of one test case is similar to that of a failed test case, it is likely to be a failed one and provide more hints on the bug location. As a result, we assign these test cases higher fitness values so that they are more likely to be chosen to generate new test cases. Similar assumption is also made in our recently published paper (Gao et al. 2015).

Per discussion presented in Section 6, when dealing with statements with same suspiciousness value, two different levels of effectiveness, namely, the best effectiveness and worst effectiveness, are utilized. However, other studies may follow different approaches to evaluate the effectiveness of a fault localization technique. For example, in (Jones and Harrold 2005), the authors used only the worst case to compare different techniques. Though reasonably conservative, we have to confess that by using only the worst effectiveness does not provide any insight on the actual effectiveness, since in most scenarios the number of statements that need to be examined is some point between the best and worst.

Then some might argue that we can instead present the arithmetic mean of the best and worst effectiveness to illustrate the actual effectiveness. However, the approach suffers from the concern that even if two techniques share the same mean of best and worst effectiveness, the actual performance may vary significantly when used in practice. Consider the following scenario: by using technique, say X, the best and worst effectiveness are five and seven, respectively, which means the bug can be located by examining six statements on average. Another technique, say Y, achieve the same level of average effectiveness whose best effectiveness is one and worst effectiveness is eleven. Can we say the two techniques perform equally in this scenario? The answer is a definite no since the variability by using X is much smaller than that of Y. To avoid such loss of information, we adopted both best and worst effectiveness in our study.

The final threat lies in the subject programs used in our study. As for studies for SPLs (Bertolino and Gnesi 2003; Burdek et al. 2015; Ensan et al. 2012; Guo et al. 2011; Mohamed Ali and Moawad 2010; Nebut et al. 2003; Perrouin et al. 2012; Perrouin et al. 2010; Reis et al. 2006) in literature, one common concern is the lack of appropriate programs available for analysis. As a result, most of the papers focus on building feature models for different SPLs. Those techniques lack in practicability and show limited applicability in industry. In a most recent paper (Burdek et al. 2015), the authors utilized only two SPLs, each of which contains less than 300 lines of code. In contrast, the smallest program used in our study contains more than 1 K lines of code, and the largest has more than 90 K line of code. Although this does not guarantee that our technique applies everywhere in industry, the empirical studies provided in this paper increase our confidence in the validity of our technique.

11 Conclusion and Future Works

In this paper, we have demonstrated the use of a genetic algorithm-based framework to generate test cases for an SPL with the integration of fault localization technique. Two different approaches, EC and KM, are presented as well as the strategy in reusing test cases from one

product to another. Our framework, which can be fully automated, can save a significant amount of time in generating test cases of good quality for SPLs.

The proposed framework was applied on four SPLs as case studies. Analysis of our results suggest that, with appropriate conversion, test cases generated in such a way can be easily reused between different products of the same family, which will help reduce the overall testing cost and provide more information from fault localization point of view. The framework which automates the process has been implemented and the quality of the test cases generated has been evaluated in terms of structure-based coverage criteria (e.g. statement, condition, all-use) and fault localization effectiveness.

In general, the feasibility of the proposed framework is sufficiently presented in this paper. Though some noise may exist in the case studies, this does not affect our conclusion that the proposed framework is effective in generating test cases of good quality for SPLs.

In the next phase, we propose to apply our framework on more SPLs to further evaluate its effectiveness. In addition, to meet the needs of companies in various areas, the methodology which can be better customized is currently being developed based on the current framework.

References

Abreu R, Zoeteweij P, Golsteijn R, Van Gemund AJC (2009) A practical evaluation of spectrum-based fault localization. J Syst Softw 82(11):1780–1792

Andrews JH, Briand LC, Labiche Y (2005) Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering (ICSE '05), pp 402–411, St. Louis, USA

Ahmed MA, Hermadi I (2008) GA-based multiple paths test data generator. Comp Operat Res 35(10):3107–3124

Ahmed ZH (2010) Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. J Biom Bioinformat 3(6):96–105

Beohar H, Varshosaz M, Mousavi MR (2016) Basic behavioral models for software product lines: expressivenss and testing pre-orders. Sci Comput Program 123:42–60

Bergey JK, Chastek G, Cohen S, Donohoe P, Jones LG, Northrop L (2010) Software product lines: report of the 2010 U.S. Army software product line workshop. Technical report. CMU/SEI-2010-TR-014. Retrieved from http://resources.sei.cmu.edu/asset_files/TechnicalReport/2010_005_001_15236.pdf

Bertolino A, Gnesi S (2003) Pluto: a test methodology for product families, 5th Interantional Workshop on Software Product-Family. Engineering 3014:181–197

Burdek J, Lochau M, Bauregger S, Holzer A, von Rhein A, Apel S, Beyer D (2015) Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines, in Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE), pp 84–99, April 11–18, 2015

Chen TY, Kuo F, Liu H, Wong WE (2013) Code coverage of adaptive random testing. IEEE Trans Reliab 62(1):226–237

Clements P, Northrop L (2001) Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston

CodeCover (2016) Available via http://codecover.org. Accessed 17 Dec 2016

Do H, Rothermel G (2006) On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans Softw Eng 32(9):733–752

Dustin E, Garrentt T, Gauf (2009) Why automate? Automated software testing ROI explained, Inform IT, April 2009

Engström E, Runeson P (2011) Software product line testing – a systematic mapping study. Informat Softw Technol 53(1):2–13

Ensan F, Bagheri E, Gašević D (2012) Evolutionary search-based test generation for software product line feature models. In: Proceedings of the 24th International Conference on Advanced Information Systems Engineering, pp 613–628, June 2012

Everitt BS (1977) The analysis of contingency tables. Chapman & Hall, Ltd., London

Fischer S, Lopez-Herrejon RR, Ramler R, Egyed A (2016) A preliminary empirical assessment of similarity for combinatorial interaction testing of software product lines. In: Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST), pp 15–18, May 16–17, 2016

Forgy EW (1965) Cluster analysis of multivariate data: efficiency versus interpretability of classifications. Biometrics 21:768–769

Freeman D (1987) Applied categorical data analysis. Marcel Dekker, Inc., New York

Gao R, Wong WE, Chen Z, Wang Y (2015) Effective software fault localization using predicated execution results. Softw Qual J. doi:10.1007/s11219-015-9295-1, online since November 11, 2015

Goel AL (1985) Software reliability models: assumptions, limitations and applicability. IEEE Trans Softw Eng 11(12):1411–1423

Goodman LA, Duncan OD (1984) The analysis of cross-classification data having ordered categories. Harvard University Press, Cambridge

Guo J, White J, Wang G, Li J, Wang Y (2011) a genetic algorithm for optimized feature selection with resource constraints in software product lines. J Syst Softw 84(12):2208–2221

Hall RJ (2015) Fundamental nonmodularity in electronic mail. Automat Softw Eng 12(1):41–79

Harman M, Jia Y, Krinke J, Langdon WB, Petke J, Zhang Y (2014) Search based software engineering for software product line engineering: a survey and directions for future work. In: Proceedings of the 18th International Software Product Line Conference, Vol. 1, pp 5–18, September 2014, Florence, Italy

Hartigan JA, Wong MA (1979) Algorithm AS 136: a K-Means clustering algorithm. Appl Stat 28(1):100–108

Henard C, Papadakis M, Perrouin G, Klein J, Traon YL (2013) Towards automated testing and fixing of re-engineered feature models. In: Proceedings of the 35th International Conference on Software Engineering, pp 1245–1248

Henard C, Papadakis M, Perrouin G, Klein J, Heymans P, Le Traon Y (2014) Bypassing the combinatorial explosion: using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans Softw Eng 40(7):650–670

Henard C, Papadakis M, Traon YL (2014) Mutation-based generation of software product line test configurations. In: Proceedings of the 6th Symposium on Search Based Software Engineering, pp 92–106, August 2014

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering, pp 273–282, Long Beach, California, USA, December, 2005

Kakarontzas G, Stamelos I, Katsaros P (2008) Product Line variability with elastic components and test-driven development. In: Proceedings of the 5th IEEE International Conference on Computational Intelligence for Modeling Control and Automation, Vienna, Austria, pp 146–151, December

Lee K, Kang KC, Lee J (2002) Concepts and guidelines of feature modeling for product line software engineering. In: Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools. ICSR-7. pp 62–77, London, UK, 2002

Liu C, Fei L, Yan X, Han J, Midkiff SP (2006) Statistical debugging: a hypothesis testing-based approach. IEEE Trans Softw Eng 32(10):831–848

Lopez-Herrejon RE, Fischer S, Ramler R, Egyed A (2015) A first systematic mapping study on combinatorial interaction testing for software product lines. In: Companion of the Eighth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2015), pp 1–10, Graz, Austria, April 13–17, 2015

MacQueen J (1967) Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. vol. 1, pp 281–297, June, 1967

Mathur AP (2008) Foundation of software testing. Addison-Wesley Professional, Indianapolis

Mohamed Ali M, Moawad R (2010) An approach for requirements based software product line testing. The 7th International Conference on Informatics and Systems, pp 1–10, March 2010

McGregor JD (2001) Testing a software product line. Technical Report, CMU/SEI-2001-TR-022, December 2001

Michael CC, McGraw G, Schatz MA (2001) Generating software test data by evolution. IEEE Trans Softw Eng 27(12):1085–1110

Mitchell M (1998) An introduction to genetic algorithms. MIT Press, Cambridge

Namin AS, Andrews JH, Labiche Y (2006) Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans Softw Eng 32(8):608–624

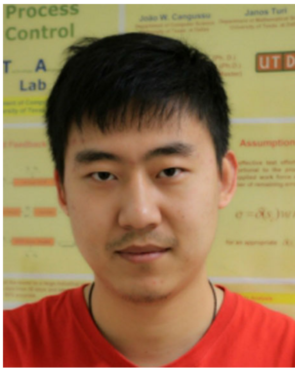
Naish L, Lee HJ, Ramamohanarao K (2011) A model for spectra-based software diagnosis. ACM Trans Softw Eng Method 20(3):11:1–11:32

Nebut C, Pickin S, Le Traon Y, Jezequel J (2003) Automated requirements-based generation of test cases for product families. In: Proceedings of 18th IEEE International Conference on Automated Software Engineering, pp 263–266, October 2003

NIST Report (2002) Software errors cost U.S. Economy $59.5 Billion Annually, NIST Planning Report 02-3, May 2002
Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Method 5(2):99–118
Perrouin G, Oster S, Sen S, Klein J, Baudry B, le Traon Y (2012) Pairwise testing for software product lines: comparison of two approaches. Softw Qual J 22(3–4):605–643
Perrouin G, Sen S, Klein J, Baudry B, le Traon Y (2010) Automated and scalable t-wise test case generation strategies for software product lines. In: Proceedings of International Conference on Software Testing, Verification and Validation, pp 459–468, Paris, France, April 2010
Pohl K, Böckle G, Linden FJ (2005) Software product line engineering: foundations, principles, and techniques. Springer, Heidelberg
Pohl K, Metzger A (2006) Software product line testing. Commun ACM 49(12):78–81
Reis S, Metzger A, Pohl K (2006) A reuse technique for performance testing of software product lines. In: Proceedings of the 11th International Software Product Line Conference, pp 5–10, Baltimore, MD, USA, August 2006
Sinha S, Dasch T, Ruf R (2011) Governance and cost reduction through multi-tier preventive performance tests in a large-scale product line development. In: Proceedings of 15th IEEE International Software Product Line Conference, pp 295–302, Munich, Germany, August 2011
Sivanandam SN, Deepa SN (2008) Introduction to genetic algorithms. Springer, Heidelberg
Telcordia Technologies (formerly Bellcore) (1998) the user's manual for the χSuds Toolsuite. Available via http://www.utdallas.edu/~ewong/SE6367/01-Project/xsuds-user-manual.pdf. Accessed 17 Dec 2016
Tijms H (2004) Understanding probability: chance rules in everyday life. Cambridge University Press, Cambridge
Vessy I (1985) Expertise in debugging computer programs: a process analysis. Int J Man Machine Stud 23(5): 459–494
Wang S, Ali S, Gotlieb A (2015) Cost-effective test suite minimization in product lines using search techniques. J Syst Softw 103:370–391
Wang S, Gotlieb A, Ali S, Liaaen M (2012) Automated selection of test cases using feature model: an industrial case study. Technical Report (2012-20), Simula Research Laboratory, 2012
Wong WE, Debroy V, Choi B (2010) A family of code coverage-based heuristics for effective fault localization. J Syst Softw 83(2):188–208
Wong WE, Debroy V, Li Y, Gao R (2012) Software fault localization using DStar (D*). In: Proceedings of The 6th IEEE International Conference on Software Security and Reliability (SERE), pp 21–30, Washington D.C., June, 2012
Wong WE, Debroy V, Li Y, Gao R (2014) The DStar method for effective software fault localization. IEEE Trans Reliab 62(4):290–308
Wong WE, Debroy V, Xu D (2012b) Towards better fault localization: a crosstab-based statistical approach. IEEE Trans Syst, Man, Cybernet − Part C 42(3):378–396
Wong WE, Mathur AP (1995a) Fault detection effectiveness of mutation and data flow testing. Softw Qual J 4(1):69–83
Wong WE, Mathur AP (1995b) Reducing the cost of mutation testing: an empirical study. J Syst Softw 31(3): 185–196
Wong WE, Qi Y, Zhao L, Cai KY (2007) Effective fault localization using code coverage. In: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC), pp 449–456, Beijing, China, July, 2007
Xie X, Chen TY, Kuo FC, Xu B (2013) A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans Softw Eng Method 22(4). doi:10.1145/2522920.2522924
Xie X, Wong WE, Chen TY, Xu B (2013b) Metamorphic slice: an application in spectrum-based fault localization. Informat Softw Technol 55(5):866–879
Xu Z, Cohen MB, Motycka W, Rothermel G (2013) Continuous test suite augmentation in software product lines. In: Proceedings of the 17th International Software Product Line Conference, pp 52–61, Tokyo, Japan, August, 2013
Yu Y, Jones JA, Harrold MJ (2008) An empirical study on the effects of test-suite reduction on fault localization. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 201–210, Leipzig, Germany, May, 2008

Xuelin Li graduated from Beihang University with a bachelor's degree in Software Reliability Engineering. He is currently on the PhD track under the supervision of Professor Eric Wong at the University of Texas at Dallas. His research interests include software testing, software fault localization, and software complex networks.

W. Eric Wong received his received his M.S. and Ph.D. in Computer Science from Purdue University. He is a full professor and the founding director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science, University of Texas at Dallas (UTD). He also has an appointment as a guest researcher with National Institute of Standards and Technology (NIST), an agency of the US Department of Commerce. Prior to joining UTD, he was with Telcordia Technologies (formerly Bellcore) as a senior research scientist and the project manager in charge of Dependable Telecom Software Development. In 2014, he was named the IEEE Reliability Society Engineer of the Year. His research focuses on helping practitioners improve the quality of software while reducing the cost of production. In particular, he is working on software testing, debugging, risk analysis/metrics, safety, and reliability. He has very strong experience developing real-life industry applications of his research results. Professor Wong is the Editor-in-Chief of IEEE Transactions on Reliability. He is also the Founding Steering Committee Chair of the IEEE International Conference on Software Quality, Reliability, and Security (QRS) and the IEEE International Workshop on Program Debugging (IWPD).



Ruizhi Gao received his Bachelor's degree in Software Engineering from Nanjing University and his Master's degree in Computer Science from the University of Texas at Dallas. He is currently a PhD student under the supervision of Professor Eric Wong, focusing on software testing, fault localization, and program debugging.

Linghuan Hu received his Bachelor's degree in Software Engineering from Chongqing University of Posts and Telecommunications. He is currently working toward his Ph.D. at the University of Texas at Dallas under the supervision of Professor Eric Wong. His research interests include software testing, symbolic execution, and Internet of Things.

Shigeru Hosono is an expert engineer at Service Business Development Division, NEC Corporation. He has been engaged in software research and development for more than 20 years since he received his M.E. in Mechanical Engineering and Applied Mathematics from Yokohama National University. He obtained his Ph.D. in Service Engineering from Tokyo Metropolitan University. His recent interests include design methodologies of service system and methods for asset-based service development. Dr. Hosono has served as chair and program committee member for IEEE Service Computing conferences (SERVICES, ICWS and SCC) since 2010. He was the chair of Technical Committee on Service Computing of IEICE (Institute of Electronics, Information and Communication Engineers) and the chair of Design and System Division of JSPE (Japan Society of Mechanical Engineers).